SIMULATION OF MANYCORE ARCHITECTURES ON MULTICORE HOSTS

by

Michael Moeng

B.S. in Computer Science, University of California, Berkeley, 2007

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Michael Moeng

It was defended on

April 14, 2015

and approved by

Rami Melhem, Department of Computer Science

Bruce Childers, Department of Computer Science

Alex K. Jones, Department of Electrical and Computer Engineering

Daniel Mossé, Department of Computer Science

Dissertation Director: Rami Melhem, Department of Computer Science

SIMULATION OF MANYCORE ARCHITECTURES ON MULTICORE HOSTS

Michael Moeng, PhD

University of Pittsburgh, 2015

Computer architects heavily rely on software simulation to evaluate new and existing processor designs. As target designs become more complex, a growing gap has emerged between single-threaded simulator performance and simulation requirements. Even though modern machines feature multiple cores, most host cores are typically unused or underutilized by state-of-the-art simulators. Parallel simulators are inherently limited by their need to synchronize threads for correctness. In my thesis, I study accurate and efficient parallelization techniques for architecture simulation.

This thesis contains several contributions. First, I study synchronization between simulator threads simulating homogeneous hardware structures such as cores or network tiles. Based on this study, I introduce a new synchronization policy, weighted-tuple synchronization, and show that it provides a better performance-accuracy trade-off compared to synchronization currently used by state-of-the-art parallel simulators. Next, I study synchronization between separate simulators responsible for modeling heterogeneous components and introduce reciprocal abstraction. Reciprocal abstraction allows asynchronous simulators to exchange information at runtime for more accurate event timing. Lastly, the reciprocal abstraction model relaxes communication latency restrictions and synchronization requirements; I show how relaxed synchronization requirements allows for coprocessor acceleration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Modeling the behavior of a processor is key to computer architecture research. The most accepted form of modeling is to simulate the architectural components of a processor, such as the cores' pipelines, memory hierarchy, and on-chip network. As simulated machines grow in transistor count and complexity, high-performance simulation is more and more difficult to achieve.

In the past, host machines improved in performance roughly as fast as target machines grew in complexity. In the multicore era, this is no longer the case—target machines are growing more complex, while the processing power of one core has plateaued. Thus, a performance gap has been growing between conventional single-threaded simulators and the computation necessary to simulate multicore target machines. Parallelizing simulators is necessary for multicore host machines to efficiently model complex architectures. However, coordinating simulator threads to maximize performance without sacrificing accuracy presents additional challenges. As simulation progress deviates between threads, they must synchronize to maintain correct event ordering. Synchronization leads to low host core utilization and reduces performance.

To address this challenge, my thesis makes the following contributions:

- Analyzes synchronization error and performs a quantitative comparison between synchronization violations and error in measured architectural metrics (such as cycles-per-instruction or packet latency).
- Introduces a novel lightweight synchronization scheme, *weighted-tuple synchronization*.
- Studies co-simulation between simulators modeling heterogeneous components, and introduces *reciprocal-abstraction* for improved accuracy via feedback-driven simulation.

- Improves performance under reciprocal abstraction by concurrently simulating asynchronous simulator components.

To synchronize simulator threads, parallel architecture simulators execute periodic barriers [52]. However, the centralized barrier scheme incurs a substantial performance overhead. A distributed synchronization policy, random-pair synchronization, was proposed to improve performance [48], but is much less accurate than barrier synchronization. After studying synchronization error resulting from incorrect simulated event orderings, also called synchronization violations, this thesis introduces *weighted-tuple synchronization*. As part of this distributed synchronization scheme, threads periodically select targets to synchronize with (match simulated times); the targets are selected heuristically in order to minimize synchronization violations. In addition, the synchronization thread granularity is generalized between the extremes of one thread and all threads; multiple threads are selected for synchronization each interval. Weighted-tuple synchronization is evaluated for two simulator settings—which model different architectures and have different thread models—and is shown to yield a superior performance-accuracy trade-off compared to both barrier and random-pair synchronization.

Often, a single simulator does not provide detailed modeling of all components a researcher wishes to study; thus, it is necessary to combine the results from two separate simulators, which is also called co-simulation. Currently, researchers rely on feeding a trace from one simulator to the other, but this approach results in significant error due to the static nature of the trace. This thesis proposes a solution, the *reciprocal abstraction* framework, where each detailed component simulator contains a simplified, abstract model of the other simulator's components; the abstract model is periodically updated for improved accuracy over the trace-drive approach. Because reciprocal abstraction allows the simulators to run asynchronously, simulator performance can be improved. The two component simulators can be run concurrently with detailed simulation of some components being offloaded to a coprocessor such as a GPU.

The three primary metrics for computer architecture simulation are speed, accuracy, and flexibility [3]. This thesis improves simulation in all three areas. The speed of simulation directly affects the time needed to evaluate new ideas. Researchers interested in architectures

with many cores or elaborate hardware structures are limited by slow simulation and cannot model complex architectures in a reasonable time frame. This thesis improves simulator utilization of a host machine's components, improving simulation speed. The distributed weighted-tuple synchronization policy improves host core utilization over the centralized barrier policy. The reciprocal abstraction framework reduces the load on the host machine by enabling the offloading of a component simulator to a coprocessor for performance gains.

Simulator accuracy is how closely the simulator software models hardware. Simulators tend to deviate from real hardware as abstractions are made for the sake of performance or flexibility. This work's study of synchronization error helps to understand the fidelity loss from parallelizing architecture simulators. The study into reciprocal abstraction identifies and fixes a major source of error caused by abstracting key latency components such as the on-chip network.

The third simulator metric is flexibility—this impacts the range of architectures a researcher can model. Often, an architect cannot find a single available simulator capable of modeling all the components he or she is interested in. Co-simulation between two component simulators can cover all components of interest, improving flexibility; however, in computer architecture simulation it is largely limited to trace-based component simulation and subject to error. The reciprocal abstraction framework allows architects to take advantage of the flexibility gained via co-simulation without sacrificing accuracy.

The thesis is structured as follows. Chapter 2 describes modern parallel simulation techniques and covers related work. Chapter 3 contains an analysis of synchronization violations which arise from loose synchronization. Chapter 4 introduces the weighted-tuple synchronization policy and evaluates weighted-tuple synchronization for parallel multicore simulation and parallel network-on-chip (NoC) simulation. Integrating simulators responsible for modeling heterogeneous components, also called co-simulation, is described in Chapter 5. Chapter 5 also describes reciprocal abstraction, a technique to loosely integrate asynchronous models for improved simulation fidelity. Performance implications of reciprocal abstraction are considered and evaluated in Chapter 6. Chapter 7 concludes and makes final remarks.

2.0 BACKGROUND AND RELATED WORK

The most widely used method of modeling a target machine's behavior is through detailed simulation, running in software on a host machine [82]. Fast and accurate simulation is crucial for computer architects to evaluate new processor designs. However, simulator performance is insufficient when simulating CMPs. For example, simulating 16 cores and their memory hierarchies using a detailed simulator will take several days [16, 80]. Research has also moved past simulating only cores and their memory hierarchy. Architects are now turning towards heterogeneous computing and Network-on-Chip (NoC) research. Adding this modeling complexity into a simulator, while retaining accuracy for conventional simulator parts (core, cache) further hinders simulation times and motivates the need for improved simulation techniques.

Core simulation refers to the simulation of a processor's core(s); the simulator models basic features such as issue width, execution order, pipeline, scoreboard, branch predictor, and register file. Often, core simulation also considers elements like the translation lookaside buffer (TLB) and L1 instruction/data caches. When simulation models more than a single core and includes components beyond the L1 caches—such as last-level shared caches, on-chip networks, memory, I/O and operating system calls—these systems become "full system simulators" [45]. Specialized simulators which target a single hardware structure are called component simulators—e.g. NoC simulators or DRAM simulators. A brief description of the simulators referenced in this work has been compiled in the Appendix, Section B.

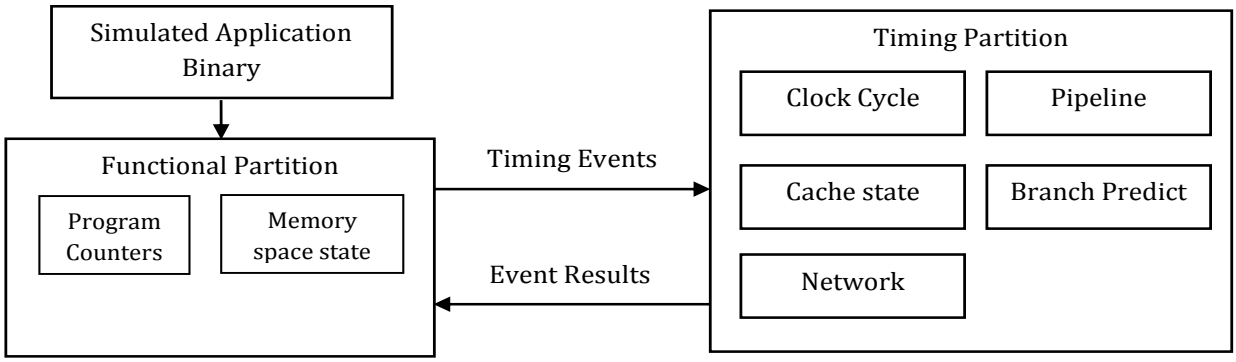


Figure 1: Functional and timing breakdown.

2.1 FUNCTIONAL AND TIMING SIMULATION

Simulation can be broken into two parts: functional and timing (shown in Fig 1). Functional simulation refers to simulation of an application binary using the target machine's ISA, dealing with control flow, system calls, and I/O requests. Timing simulation refers to modeling the timing of any hardware structures of interest to an architect. These can be core components such as a pipeline, or uncore components such as an on-chip network. The amount of simulated time these structures take to process data or requests is used to determine the target machine's performance.

Much of the simulation state used in functional simulation, most notably data values, are not used during timing simulation. Similarly, much information obtained during timing—such as specific cache tags—are not needed by the functional simulator. This clean division of data results in many simulators being developed with a functional/timing partition, such as in [9, 14–16, 48]. The functional partition handles functional simulation and generates events needed by the timing partition. The timing partition calculates how long these events take to resolve and notifies the functional partition if it needs to deviate from regular functional simulation (such as simulating the wrong path of a branch).

In some cases, the functional portion of simulation is represented by a static trace—this is also known as trace-driven simulation [75]. Trace-driven simulation is commonly used for specialized component simulators when generating traffic is time-consuming—using one trace helps to amortize the traffic generation time. In this case, inaccuracies can arise if the

functional and timing components disagree on which instructions should be executed. This can occur due to branch mispredictions [15], or from incorrect ordering for race conditions in multi-threaded programs [16, 50]. In order to decode the correct instructions, the functional simulation requires feedback from the timing model.

Instead of decoding a program binary and tracking register/memory state in software, it is also possible to run the program natively on the host machine, then get relevant functional information via instrumentation; this technique is known as native simulation or emulation. For native simulation, the program transitions from functional to timing simulation whenever it reaches instrumentation code. Native simulation can vastly improve performance, as it almost completely eliminates time spent for functional simulation. Simulators which use native simulation use a binary instrumentation/translation frontend. Binary translators for X86 such as QEMU [4] and PIN [44] are used to generate code for core simulators such as MARSS [58] (QEMU), Sniper [9] (PIN), or ZSim [67] (PIN). The Ocelot PTX emulator [20], dynamically translates PTX code for NVIDIA GPUs and is used to feed GPU traffic to the heterogeneous MACSim simulator [37].

2.2 SIMULATION SAMPLING

Architects use sampling techniques to run shorter simulations in a way that is still representative of machine behavior. This occurs by either reducing the target machine's workload or by only simulating portions of a full workload in detail.

2.2.1 Shortened Workloads

Target machines are typically evaluated while executing a workload or benchmark program. Full benchmarks, which are designed to evaluate real machine performance, are often executed on simulated machines [6, 71, 79]. With native inputs, these benchmarks take too long to run on simulated machine; architects tend to either use smaller input sets or entirely new benchmark suites reworked for shorter but still representative execution [38, 73]. Another

approach is to use entirely synthetic workloads. These can either be simple traffic patterns or statistically generated workloads; statistical simulation profiles a longer-running benchmark and produces a shorter synthetic trace which has similar execution characteristics [21,54,56].

2.2.2 Sampled workloads

Another form of sampling is to only model portions of the program in full detail, as seen in the SMARTS framework [78,80]. Initially, the simulator will only perform functional simulation to fast forward to a point of interest. It then performs detailed modeling without gathering statistics to warm up structures such as the cache or branch predictors. Finally, the detailed simulation occurs for a period before the simulator repeats the process for the next selected program region. Simpoint [69] proposes to statically analyze the program to select the most representative regions for detailed simulation. In order to reduce the time spent for repeat runs on fast forwarding and warmup, the simulator state can be checkpointed [77]. The target program is simulated once through fast forwarding and warmup, then all simulation state is saved. The resulting checkpoint can then be used for separate configurations to amortize the fast forwarding and warmup cost.

2.3 MODEL ABSTRACTION

When a simulator exactly models the cycle-level behavior of a hardware component, it is performing detailed simulation of the component. However, it is not feasible for a simulator to model every hardware component in a machine perfectly. Some hardware components are modeled abstractly or approximately for improved performance. Out-of-order cores and uncore/off-chip components are popular targets for abstraction due to their complexity and simulation requirements. Abstractions can be performed mechanistically or empirically [28]. A mechanistic model is a simplified model of the target, constructed using knowledge of the intended behavior [24,35,70]. An empirical model is constructed without knowledge of the target system, but is trained to approximate the target's behavior [32,39,40].

2.3.1 Simplified Core Models

Detailed modeling of an out-of-order core is very costly; thus some simulators simplify the out-of-order behavior. Interval simulation models the core's performance in terms of long-latency events such as cache misses, TLB misses, or branch mispredictions [28]. Similarly, In-N-Out uses memory access dependencies to approximate the reorder buffer state from a functional model with in-order instructions [41]. Most simplified out-of-order core pipeline models process instructions one at a time, in the same manner as in-order core simulation. Consequently, an instruction's latency components are determined atomically (e.g., all latencies calculated at the time of instruction issue). Atomic instruction processing is widely used by simulators which rely on binary instrumentation such as Graphite [48], Sniper [9], or ZSim [67].

In many full-system architecture simulators, specialized uncore or off-chip components often use an abstract, simplified model for behavior. For example, network packets, main memory accesses, and I/O requests may use a fixed latency or simple queue model. These simplified components are used in many simulators, but are mandatory for simulators with atomic instruction processing because detailed component simulation requires multiple outstanding requests for accurate contention modeling. The goal of these abstract component models is to immediately estimate the latency of a timing event so the core model can determine the instruction's latency before simulating the next instruction.

2.3.2 Queue Models

Abstract component models break down latency based on the hardware structures a request traverses; these are known at the time of instruction issue. For a main memory or I/O request, there may only be a single structure. For a network packet, each router hop between the source and destination is a structure that must be modeled. Each hardware structure in the component model is represented using an abstract model. The most basic model assumes no contention; a request's latency assumes there were no contending requests.

More complex queue-based models can be used; for example, M/M/1 and M/G/1 queues were evaluated for network architectures in [55]. A particular queue model available in

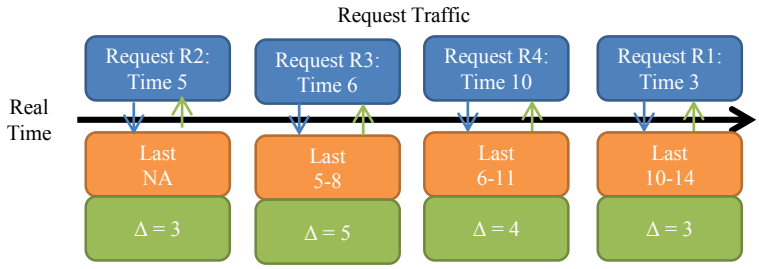


Figure 2: Illustration of a simple contention queue model where requests occupy the hardware structure for 3 cycles. The structure sees four requests arrive at simulated times 3, 5, 6, and 10. Their real-time arrival order is R2 → R3 → R4 → R1.

the Sniper full system simulator that performed well empirically compared to cycle-level simulation is its "simple contention" model. The contention queue model is illustrated in Fig. 2. The queue tracks the time of the last request, which reserves Δ cycles, where Δ is the base request latency. If another request arrives during these Δ cycles, it must wait for the first request to clear the queue. For example, in Fig. 2, request R3 arriving at simulated time 6 must wait for R2 to clear the queue at time 8; R3 then occupies the hardware component for cycles 9-12. Because parallel simulators are loosely synchronized, requests can occur out of order relative to how they would in a sequential modeled system. Requests which arrive prior to the last packet are assumed to encounter no contention, such as request R1 from Fig. 2.

2.3.3 Abstract Memory Hierarchy

Some works study an abstract model of the cache hierarchy. In StatCache [5], authors feed statistics from one sample simulation run into an abstract cache model—the abstract model can then estimate cache behavior for a wide range of cache configurations without further simulation. Its abstract model tracks the reuse distance to a set of sampled memory addresses to determine the working set size, which then can be used to estimate miss rates for caches of varying size. The StatCache model is limited to fully-associative caches with a random replacement policy. A later work, StatStack [22], targets LRU replacement. A sample of memory accesses is taken, with reuse distances gathered as with StatCache. However, the

probability of repeat accesses to addresses within an interval is also profiled. Combined with the reuse distance, a cache's LRU performance can be estimated.

2.3.4 Feedback-driven Abstract Models

Prior works have utilized timing feedback with the goal of accelerating simulation. Speedups are obtained by periodically disabling a detailed part of the simulator's timing model and relying on a simplified model. The simplified abstract model relies on feedback from the timing model to maintain accuracy. With these schemes, the detailed model is not always active and cannot provide full statistics. These schemes resemble simulator sampling techniques described in Section 2.2.2.

In COTSon [2], authors propose *functional-directed* simulation. An emulator manages the functional portion of simulation while a detailed simulator determines timing; the IPC from the detailed simulator is fed back to the functional emulator to control the relative progress of each thread. Because the detailed timing model dominates simulation time, speedups are achieved by taking samples from the detailed model to control functional IPC. For periods where the detailed simulator is inactive, the most up-to-date IPC serves as a simple performance model for the entire timing simulator.

Another work, FIST [57], aims to accelerate multicore simulation by replacing a detailed network model with an abstract model. The authors represent network routers abstractly as load-latency curves—when a packet passes through the router the number of recently-traversed flits, representing the router's load, is used to determine the latency. In practice, the load-latency curve is represented as a table indexed by load. This model can be trained offline or online with a detailed network model; online training is similar to feedback-driven simulation. As with COTSon [2], speedups are achieved by disabling the detailed network for periods of time and relying only on the load-latency curves.

In this thesis, timing feedback improves accuracy by integrating two simulators in a situation where full integration simulation is not feasible due to synchronization issues. In part because direct integration is not possible, the information exchange between the two models is imperfect. For example, network load cannot be precisely calculated by the abstract

model. Another difference is that all portions of the simulator remain active in order to obtain detailed statistics; unlike prior approaches, the simulator does not switch between detailed and abstract mode.

2.4 PARALLEL SIMULATION

To close the performance gap from single-threaded simulation, researchers investigated parallelizing simulators. In a parallel simulator, simulated components (such as cores) are divided among simulator threads; each thread models the components assigned to it. For example, when simulating a multicore machine, each simulation thread might model one or more cores. While parallel simulation can improve performance, it also introduces new challenges. A major factor limiting the performance and accuracy of parallel simulators is simulator thread synchronization.

To study the need for synchronization, it is necessary to understand a simulator's notion of time. Target machines have their own simulated time, measured in cycles, which is monotonically increasing. Time advancement varies between simulators. Time-stepped simulators, such as Hornet [43] have unit cycle advancement. Each cycle, the behavior or individual simulated components is simulated before moving to the next cycle. Event-driven simulators, such as GEM5 [7], simulate events that occur in the machine, and ignore hardware structures which do not have scheduled events. In event-driven simulation, cycles with no events scheduled are skipped to save time. As a result, event-driven simulators generally advance time in multi-unit steps. It is possible for events to be scheduled every cycle, as is the case for GEM5 in detailed mode, in which case event-driven simulation and time-stepped simulation have the same time advancement. However, some event-driven simulators do not schedule events every cycle; for example, Sniper [9] is an event-driven simulator whose events correspond to simulated instructions. In this case, time advances unevenly based on the simulated instruction.

In parallel simulators, each thread has its own simulated time, which represents the minimum cycle time of all components modeled by the thread. While a thread can ensure

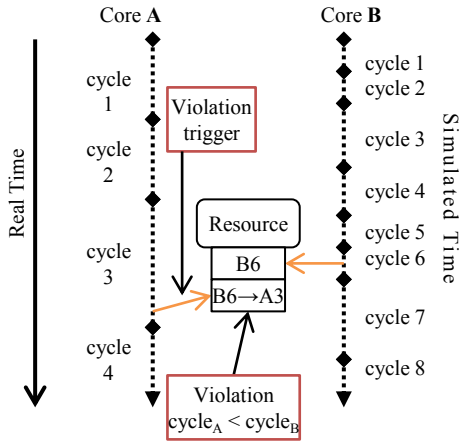


Figure 3: Synchronization Violation.

its components have the same simulated cycle, it cannot make guarantees about the clocks of other components. Threads advance time at different rates due to workload imbalance and host OS scheduling artifacts. Thus, over the course of simulation, simulated clocks deviate. When threads with different clocks interact by exchanging messages or accessing shared resources, the timing of these events can occur in an incorrect order than they would have in a serial simulator, causing a violation of the simulation state.

Violations were previously defined in [13]. Figure 3 illustrates a violation caused from out-of-order accesses to a hardware resource. At cycle 6, Core ***B*** accesses the resource. Because cores are not kept perfectly synchronized, Core ***A*** accesses the same resource at cycle 3, triggering a synchronization violation. After the violation occurs, the hardware resource's state incorrectly reflects an access by ***B*** in cycle 6, followed by an access by ***A*** in cycle 3. In this example, Core ***A***, which made the first access, is ahead in real time but behind in simulation time. Core ***B***, whose access triggers the synchronization violation, is behind core ***A*** in real time but ahead in simulation time.

2.4.1 Synchronization Techniques

There are two extremes when considering parallel simulator synchronization: *unbounded* and *cycle-by-cycle*. With unbounded simulation, no synchronization is employed and cores are simulated at whatever rate the host machine allows; both performance and synchroniza-

tion error are maximized. Using cycle-by-cycle synchronization means that all cores are synchronized every cycle with a barrier.

Barrier synchronization, also called *quantum* synchronization, is a more general form of cycle-by-cycle synchronization described in [52]. Every N cycles—the quantum length—the simulator executes a barrier. Synchronization with an N-cycle quantum is called *barrier*-N; cycle-by-cycle synchronization is equivalent to *barrier*-1. An architect can control accuracy by varying the quantum length: a larger quantum results in higher performance but lower accuracy.

Slack synchronization was proposed as an alternative to barrier synchronization [12,13]. Instead of using barriers, threads stay synchronized using a global time (defined as the lowest clock count among all cores). All cores keep their clocks within a certain number of cycles—the slack—from the global time. For slack simulation, the slack size serves the same purpose as the quantum size does for barrier synchronization, controlling the trade-off between accuracy and performance.

A distributed, lightweight alternative to global synchronization schemes such as barrier and slack synchronization is *random-pair* synchronization. This synchronization scheme is implemented by the Graphite simulator [48]. Under random-pair synchronization, simulated core $\boldsymbol{A}$ periodically selects another core at random, core $\boldsymbol{B}$. If $\boldsymbol{A}$ has simulated more cycles, it attempts to synchronize with $\boldsymbol{B}$ by putting itself to sleep until $\boldsymbol{B}$ catches up. The overall simulation rate and the difference in clocks between the two cores are used to estimate the sleep time. As is the case with barrier synchronization, the quantum between synchronization attempts controls the performance-accuracy trade-off for random-pair synchronization.

The ZSim simulator uses a two-phase bound-weave synchronization strategy [67]. During the bound phase, barrier synchronization is enforced, and a trace of L1 misses is generated. During the weave phase, trace items are processed using conservative Parallel Discrete Event Simulation (PDES). Because processing of events depends on the trace generated in the bound phase, bound-weave synchronization cannot avoid error when the program's control flow or data access pattern are nondeterministic. The authors of ZSim do not analyze two-phase bound-weave synchronization except to show scalability with host core count. It is unclear how much performance overhead two-phase bound-weave synchronization incurs on

top of the barrier synchronization used during the bound phase; or how accuracy compares with other synchronization techniques.

Some program phases are more prone to synchronization violations than others. In order to maintain a consistent violation rate, it is possible to adaptively adjust the synchronization quantum. Dynamically adapting the synchronization interval for barrier synchronization is studied in [25]. The synchronization quantum is adjusted based on the number of network packets traversing the network. In [12], authors apply adaptive synchronization to slack simulation. First, a target violation rate is specified. During the course of simulation, the synchronization quantum is dynamically adjusted to reach the target violation rate.

Weighted-tuple synchronization, presented in this work and published in [51], is a decentralized scheme which targets CMP hosts. While prior schemes consider only extremes of thread synchronization granularity (all threads or one thread), weighted-tuple synchronization uses a middle-ground approach—experiments show synchronizing with four to eight threads provided significantly better accuracy with minor performance penalty. Further accuracy is obtained by selecting targets in order to reduce state violations. In addition, weighted-tuple synchronization is orthogonal to adaptive synchronization, as weighted-tuple relies on an adjustable synchronization quantum; thus, both policies can be combined for additional benefit.

Adaptive Synchronization:

2.4.2 The Sniper Multicore Simulator

The Sniper [9] and Hornet [43] simulators are referenced heavily in this work and are used to evaluate both weighted-tuple synchronization and reciprocal abstraction; the simulators are described here in detail. Sniper was introduced as an extension to the Graphite simulator [48]. Like Graphite, Sniper uses PIN [44] to dynamically translate x86 binaries for functional simulation. Because binary translation leads to the simulator processing instructions one at a time, it is not straightforward to model an out-of-order processor with multiple outstanding instructions. Sniper alleviates this issue by using an interval simulation to abstractly model out-of-order core pipelines [28].

Because the program is being passed through PIN, Sniper launches one simulation thread for each modeled thread. Most often, this means one thread is responsible for modeling each target core. Sniper relies on barrier synchronization to keep maintain correct thread progress. When a core accesses remote state (such as a remote cache), the thread responsible for the accessing core directly accesses the requested state through shared memory.

In addition to modeling out-of-order cores, Sniper models a cache hierarchy with a directory-based coherence protocol. Abstract models are used to calculate latencies for traffic between cores and latencies for DRAM accesses. These latencies are estimated using queues or reservation windows as described in Sec. 2.3.2.

2.4.3 The Hornet Network-on-Chip Simulator

The Hornet simulator models a network-on-chip architecture at a cycle level. An example using a mesh topology is shown in Figure 4. Each tile's home node injects packets, which are converted into flits, consisting of a head flit and several body flits. At each router, the head flit computes its next hop according to the routing policy, then allocates a virtual channel in the next router. Once this is performed, the head flit and all body flits contend for the router crossbar switch and traverse the link to the next router's virtual channel buffers. This process is repeated for each hop until the packet reaches its destination. Simulation of network tiles is broken into the positive clock edge and negative clock edge—route computation, virtual channel allocation, and switch allocation/traversal occurs in the positive edge, while link traversal occurs in the negative edge.

Hornet is a parallel simulator, which partitions tiles evenly among threads; for example, with 256 network tiles and 32 host threads, each thread simulates 8 tiles. Tiles modeled by the same thread are synchronized relative to one another; Hornet uses barrier synchronization to keep tiles from different threads synchronized. Flits are modeled in detail, thus they are passed off from tile to tile. This means a flit is often handled by two or more threads as it travels to the destination.

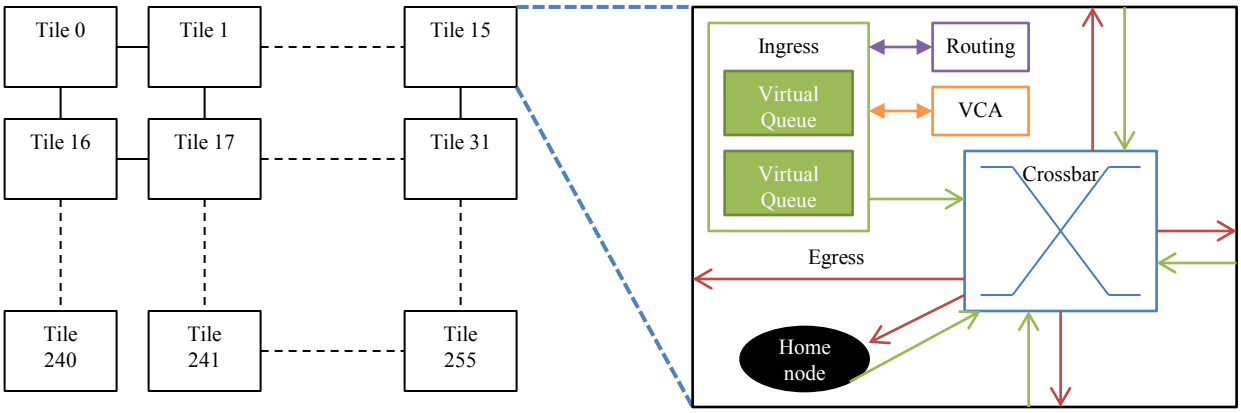


Figure 4: Network tile architecture with an example 2-D mesh topology. Each tile contains an ingress and egress for each link and for the home node. An ingress consists of multiple virtual queues corresponding to virtual channel buffers. Egresses are linked to a corresponding ingress in the connecting tile. Tiles also models structures which handle routing, virtual channel allocation (VCA), and switch allocation/traversal (crossbar).

2.4.4 Parallel Discrete Event Simulation

A widely studied parallel simulation approach for general fields is Parallel Discrete Event Simulation (PDES); overviews can be found in [26,61]. When a model contains a sequence of discrete events, executing on separate logical processes (LPs), these events can be simulated in parallel while maintaining correct event ordering. Event orderings can be maintained conservatively or optimistically.

Under conservative PDES, an event only processed when it is guaranteed no earlier event will arrive, using a *lookahead* value [10, 11]. The lookahead is defined as the amount of simulated time a thread can safely simulate without risking incorrect event orderings. Optimistic PDES allows threads to potentially generate incorrect event orderings, but rolls back the simulation if a violation is detected according to the time warp algorithm [18, 33]. First the thread which detects a time violation must roll back its state. It then must also cancel all messages sent out during rollback, starting a cascading rollback until state is correct. Once all incorrect events are cancelled, simulation resumes.

Parallel simulation of CMP hosts using PDES techniques is described in [14]. The communication latency between cores is designated the lookahead value. By setting a quantum size less than the lookahead, the simulator can guarantee that events will cause state vio-

lations. The authors recognize that simulating distributed L2 slices can be very costly. In computer hardware, events tend to occur every cycle on every core; especially with more detailed modeling of pipelines or network-on-chip routers, which leads to a lookahead value of zero. This makes fully-accurate PDES techniques less efficient for simulating CMPs. The SST [65] simulation framework and the Manifold [76] simulator synchronize timing using conservative PDES. The Manifold simulator groups network tile simulation into one logical process; creating a non-zero lookahead equal to the time it takes for an L1 access to become miss and be injected into the network. For improved performance, Manifold also supports barrier synchronization.

While most PDES works assume fully-accurate synchronization, some research has experimented with relaxed ordering constraints. Relaxing event ordering constraints can allow PDES to be efficient in settings with zero lookahead. Rather than enforcing total event ordering, alternative orderings are proposed. In [64, 72], unsynchronized PDES is evaluated for a network of queues. It is shown to be most accurate when modeling a stable, low-traffic system.

Approximate-Time (AT) ordering allows events to be processed within a time range, and is evaluated in [27, 47]. Barrier synchronization is a policy which enforces AT ordering, where each event's time interval is equivalent to the synchronization quantum. In [27], approximate-time causal ordering is also proposed, which strengthens AT ordering by adding causality restraints. Distributed synchronization schemes, used in this work, are more relaxed than approximate time orderings as no guarantees are placed on event processing time; however, this work's evaluation of weighted-tuple synchronization shows that it is as accurate as the AT-ordered barrier synchronization policy.

Porras et al. [62] group LPs into subsets or neighborhoods based on distance; LPs in the same neighborhood are synchronized conservatively. By having each LP only synchronize with a subset of other LPs, the PDES algorithm can more efficiently take advantage of lookahead for improved parallel performance. Messages to a different neighborhood may arrive at the wrong time and cause violations. The authors' region-based synchronization is evaluated for simulation of a wireless phone network and the accuracy loss is determined to be minor. The concept of regional synchronization is similar to this work's weighted-

tuple synchronization in that each process synchronizes with the subset of processes it is most likely to communicate with. However, under weighted-tuple synchronization the tuples are re-selected each synchronization interval as opposed to the static neighborhoods. This makes weighted-tuple a more accurate synchronization scheme when traffic can cross regional boundaries, as is the case for both multicore and network-on-chip simulation.

2.5 INTEGRATING MULTIPLE SIMULATORS

While a great deal of computer architecture simulation focuses on the behavior of the cores and memory hierarchy, some research is focused on the behavior of specific components such as the NoC or main memory, which are not modeled in detail by most core simulators. A simulator which only models a specific component still requires realistic traffic as input. A widely-used traffic generation approach is to use a core simulator to generate traffic as a trace, then feed the trace into a specialized component simulator. However, the impact of architectural changes to the specific component will not be reflected in the static trace. In a real machine, longer latencies in the NoC or main memory would lead to lower core throughput and slower component traffic.

In order to accurately model multiple components, integration is required. Serial, cycle-accurate simulators such as M5 [8] and Garnet [1] can be directly integrated [7], but are too slow when considering hundreds of cores in a target system. Integration with parallel simulators is not straightforward, particularly when the simulators advance time differently or when they use different synchronization granularities

First, when simulators advance time differently, they cannot synchronize with one another directly; an event-driven simulator with multi-cycle time advancement abstracts events a time-stepped simulator needs to model in detail. For example, a core simulator may treat an outgoing memory request latency as the sum of the cache, network, and main memory latencies. However, a detailed component simulator needs to model multiple events such as individual router latencies or main memory commands before determining the final latency.

Second, parallel simulators have acceptable accuracy at different synchronization intervals. For example, core simulation uses relatively loose synchronization [13, 52], primarily because the communication time between cores is, at minimum, the L1 access latency plus the L2 access latency plus network travel time. For a NoC simulator, however, flits typically traverse links in a single cycle. In this setting, finer-grained synchronization is necessary for parallel NoC simulators [43]. Integration between the two models would require them to use the same synchronization interval, which would lead to either a drop in performance for the coarsely-synchronized simulator or a drop in accuracy for the finely-synchronized simulator.

The current techniques used to model specific components with a core model are trace-driven simulation and single-threaded, fully-integrated simulation; these are, respectively, inaccurate and slow. This work proposes reciprocal abstraction, a co-simulation technique which integrates parallel simulators while maintaining accuracy. Furthermore, while parallel simulators have limited scalability, concurrent execution with another parallel simulator can improve overall host resource utilization.

Co-simulation is a general term used to describe integration of separate simulators for concurrent execution and is used in diverse settings such as coupling heat/airflow models or power grid/communication network models [19, 30, 74]. The key challenges for co-simulation are communication and synchronization between simulators. Co-simulation most often forces tight coupling, where simulators are kept synchronous. In [29], a loose coupling approach was proposed, where simulators alternate time steps and the output from each time step is fed to the next simulator. Reciprocal abstraction resembles this approach, although the core timing simulator passes exact data (individual packets) to the network simulator rather than aggregate data as used in the loose coupling co-simulation approach.

2.6 COPROCESSOR ACCELERATED SIMULATION

One approach to deal with increasing simulation complexity is to simulate hardware component(s) on a specialized coprocessor. Because the coprocessor is specialized in only a few tasks and is more difficult to program than general-purpose processors, generally only a lim-

ited part of the architecture is simulated on the coprocessor. The host's CPU handles the rest of the simulation. One challenge researchers must deal with when using a coprocessor is the long communication latency between the host CPU and the coprocessor.

Most prior work attempting to use a coprocessor to accelerate computer architecture simulation has focused on Field-Programmable Gate Arrays (FPGAs). The RAMP project [59] was started in order to use FPGAs for development of manycore systems. Chiou et al. [15] argue that fine-grain parallelism is necessary to accelerate cycle-accurate simulations, and implement a detailed timing model on an FPGA while the functional model runs on a CPU. A host CPU takes care of functional simulation, and is rolled back when a conflict is reported by the timing model. HAsim [60] deals with space constraints on an FPGA by time-multiplexing simulated cores and NoC routers. After time-multiplexing, simulation rates are limited by on-board memory capacity. Time-multiplexing is automatically performed by GPUs with enough threads, but techniques to deal with long latency communication to a host CPU and to reduce a simulator's memory footprint are directly applicable to GPU-based simulation. A downside of coprocessors is they are generally difficult to program; the ACME tool automatically generates VHDL code for an FPGA based on a graphical description [31].

GPUs are an emerging platform for accelerating computer architecture research, especially for target systems with hundreds of homogeneous cores. Raghav et al. proposed using GPUs to do functional simulation for manycore architectures [63]. They parallelize the processes of decode and lookup, but instruction execution suffers from warp divergences if the instruction mix is heterogeneous. In [49], authors simulate a large number of L1 and L2 caches on a GPU. Each cache way is mapped to a CUDA thread, which allows parallel tag lookups. Cache traffic is trace-driven; the authors use software pipelining to overlap trace I/O by the host CPU with kernels simulating the L1 and L2 caches. Their work synchronizes caches only every kernel invocation, each covering a couple thousand accesses. Consequently, error is introduced when caches on separate blocks communicate.

This work utilizes GPUs to accelerate simulation, although the general technique can be applied to other coprocessors. Under reciprocal abstraction, only a portion of simulation is offloaded. The partitioning of work between the host CPU and GPU is described further

in Chapter 6. By offloading a portion of simulation, the host cores have less work; thus, performance of the coprocessor-based simulator is not as critical.

3.0 ANALYSIS OF ARCHITECTURE SYNCHRONIZATION

This chapter studies parallel simulation and synchronization error which occurs due to loose synchronization. Various methods of measuring synchronization error are discussed; the synchronization violation rate in particular is motivated as a synchronization error metric which is accurate and easy to measure. The initial proposal for random-pair synchronization targeted distributed hosts with long communication latencies; before showing experimental results, this chapter described and evaluates a CMP-optimized implementation. The random-pair implementation introduced in this work is more precise at shorter synchronization intervals.

Barrier and random-pair synchronization are studied in two simulation settings on a CMP host. The first setting is multicore simulation; the error trends for barrier and random-pair are analyzed at varying synchronization intervals. In addition, the use of violations as an error metric is motivated in two ways. First, the violation rate is shown to be correlated with architectural metric error. Second, the instability of architectural metric error, specifically cycles-per-instruction (CPI) error, is explained and demonstrated.

The second simulation setting evaluated is parallel network-on-chip simulation. Error trends with varying synchronization quanta are again analyzed; the analysis shows that performance for random-pair synchronization is similar for all synchronization quanta, but is more accurate at shorter intervals. Error is represented in terms of violation rate and network latency with similar trends observed.

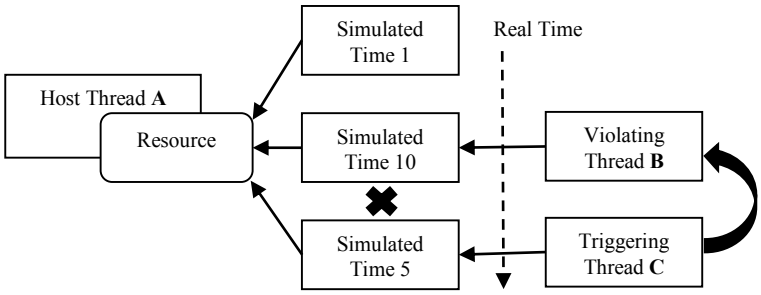


Figure 5: Definitions for tracking violations.

3.1 SYNCHRONIZATION ERROR

This section motivates the use of synchronization violations when measuring synchronization error. It defines violations in detail and describes their measurement. Other synchronization error metrics have been proposed and used in the past, such as architectural metric error and clock skew. These alternate metrics are described and compared with violations.

When simulation time deviates between threads, accesses to simulated hardware structures are initially *speculative* and are time-stamped with the simulation time. An access made to a resource after a speculative access either has an earlier timestamp or a later timestamp. If the second access has a later timestamp, there was no violation. However, if the second access carries an earlier timestamp, a synchronization violation has been triggered; the first speculative access was issued too early and should have occurred after the second access.

Figure 5 illustrates a violation and the threads involved. The resource being contended over is local to the *host thread*, ***A***, but thread ***A*** is not involved in this violation. Thread ***B*** was responsible for the incorrect speculative access at time 10 and is the *violating thread*. Thread ***C***'s access actually discovers the violation, making thread ***C*** the *triggering thread*. Upon triggering the violation, thread ***C*** notifies thread ***B*** of the incorrect speculation, and that ***C*** was the trigger. Notification can occur either sending a message or by directly modifying thread ***B***'s shared memory.

Because there are many hardware resources, it is most practical to sample violations by monitoring a particular hardware resource [13]. A sampled resource is reasonable as long as

it gives good coverage of violations likely to affect simulation correctness. In the multicore simulation setting, this work monitors cache sets for coverage of memory access and coherence violations. Only remote accesses are considered for violations, as grouped local accesses make up the vast majority of accesses and cannot cause violations. In the NoC simulation setting, traffic traversing crossbars in the switch traversal phase is monitored, which covers all network traffic.

A widely-used approach to measure synchronization error is to compare architectural metrics, especially performance metrics, between loosely-synchronized simulation and a fully-synchronized baseline. For example, in the network simulator Hornet [43], authors measure synchronization accuracy by comparing the average network latency, using single-cycle barrier synchronization as the baseline. In [9] and [67], authors compare architectural metrics, such as program execution time, between those measured by the simulator and those measured on a real machine with a matching configuration.

In the setting of relaxed PDES (Section 2.4.4), Fujimoto measures the percentage of events which are processed at the correct time [27]. This metric is similar to the violation rate, as a violation can only occur when at least one event is processed at the wrong time. However, in a low-traffic scenario, a simulation could have incorrectly timed events with no violations. In this situation, simulation state is correctly maintained. Fujimoto also measures the architectural metrics of average job waiting time and job throughput, as do authors of [64, 72].

Violations have a distinct advantage over architectural metrics for error measurement in that no baseline is required. Generating an error-free baseline simulation using a fully-synchronized simulation run is infeasible if a researcher wants to take advantage of speedups provided by a parallel simulator. One alternative to a fully-synchronized baseline explored in the past is using a real machine as a baseline [9, 67]. A real machine may not exactly match a target machine configuration; moreover, real machines cannot match futuristic target machines with a larger number of cores. Another alternative is to use lax barrier synchronization as a baseline; for example, in [48] a 1000-cycle barrier interval is used as a baseline. This is undesirable as the loosely-synchronized baseline has synchronization error which is unknown.

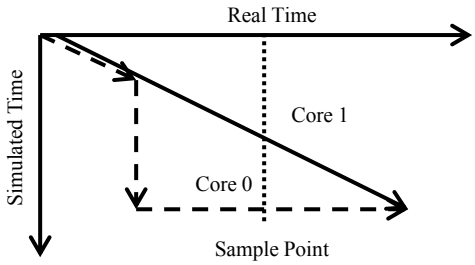


Figure 6: Example of False Skew.

Measuring violations allows accurate error measurement without a costly baseline or access to a real machine with a matching configuration.

A second issue with architectural metrics is they are subject to an averaging effect. Because they have a non-zero baseline, program phases where the metric is too high or too low will average out, leading to unstable error trends where looser synchronization appears to have lower error. A more comprehensive study of the averaging effect is presented in Section 3.3.4. In addition, correlation studies between violations and architectural metric error are presented in Sections 3.3 and 3.4.

The Graphite simulator measures error as average clock skew [48]. The simulator periodically samples the current clock cycle for all simulated cores and the total skew is summed up. This approach will yield some false errors, because it attempts to sample the core states kept by asynchronous threads. A core can increase its time stamp but not make any incorrect accesses. Figure 6 illustrates a case where the thread simulating core 0 encounters a long latency stall, but the thread correctly waits for core 1 before accessing other resources. A clock skew sample, taken during the waiting period, would detect a large skew even though no violations occur.

In [67], authors define accesses which cause path-altering interference. Two accesses incur path-altering interference if a change in their ordering changes their paths through the memory hierarchy. Their categorization of accesses that cause path-altering interference is similar to synchronization violations used in this work and in [13]. Violation tracking used in this work differs from path-altering interference in several ways. Local accesses are counted for accesses that may cause path-altering interference. Most core accesses are local and will not be reordered relative to one another, which greatly reduces the measured percentage of

accesses with path-altering interference. In addition, two hits are never counted for path-altering interference. Two hits with incorrect ordering still constitute a violation for caches because the ordering still affects the replacement policy state and can indirectly alter the path of other accesses through the memory hierarchy.

3.1.1 Impact of Synchronization Error

Synchronization error causes simulation measurements to deviate from fully-synchronized simulation results. If not enough runs are executed, lack of synchronization can lead to incorrect conclusions. The correct methodology with non-deterministic simulation is to execute multiple simulation runs to ensure that the results fall within a confidence interval. The additional runs required due to synchronization violations may negate the speedup achieved from looser synchronization. This section performs a study on the number of additional runs necessary to reach a stable result. A multicore simulation setting was used for evaluation, simulation configuration details can be found in Sec 3.3.1.

Given the standard deviation σ, the number of runs N, and the Z-value corresponding to a specified confidence percentage, a confidence interval I can be calculated as the deviation from the mean further experimental results are likely to fall within. By manipulating this equation to the form shown in Eq. 3.1, and inputting the remaining values, the number of runs necessary to achieve a specified confidence interval can be found.

$$N = \left(\frac{Z_{95\%} \times \sigma}{I_{5\%}}\right)^2 \tag{3.1}$$

The target metric for this study was performance, measured as cycles-per-instruction or CPI. The standard deviation, σ, is the standard deviation the CPI from three runs for each synchronization quantum (1, 10, 100, and 1000 cycles). This sample standard deviation did not change significantly after repeating three additional runs, showing that the three-run sample is sufficient to represent the standard deviation for each particular benchmark/synchronization interval pairing. The value for Z was found by looking up the Z-value corresponding to the specified confidence percentage for a standard distribution (1.96 for 95%). The value for I was calculated by multiplying the mean for each benchmark's CPI

Table 1: Required Runs for Confidence

Benchmark	*barrier*-1	*barrier*-10	*barrier*-100	*barrier*-1000
barnes	4	3	1	183
blackscholes	10	15	11	397
canneal	1	2	2	438
lu.ncont	2	11	54	209
ocean.cont	1	1	5	10
water.nsq	9	49	45	424
geomean	3	7	9	175

by 5%. The number of runs required to obtain the desired confidence interval was then estimated; results are listed in Table 1.

The number of required runs generally increases with less accurate synchronization, especially for *barrier*-1000. Depending on the accuracy requirements, however, a smaller or larger confidence interval may be tolerable, which would affect the number of required runs per configuration. One trend of interest is that the number of required runs is not monotonically increasing as the synchronization becomes less frequent. In Sec. 3.3.4, CPI error is shown to be subject to an averaging effect. The averaging effect can cause runs with less synchronization to appear more accurate despite having more synchronization violations.

The accuracy loss from loose synchronization results in more required simulations; Figure 7 shows the adjusted simulation time necessary to obtain the desired confidence interval: the simulation time for a single run is multiplied by the number of required runs from Table 1. For the 5% confidence interval with 95% confidence, *barrier*-100 requires the lowest average simulation time. Overall, this study motivates the need for a synchronization policy which is both fast and accurate.

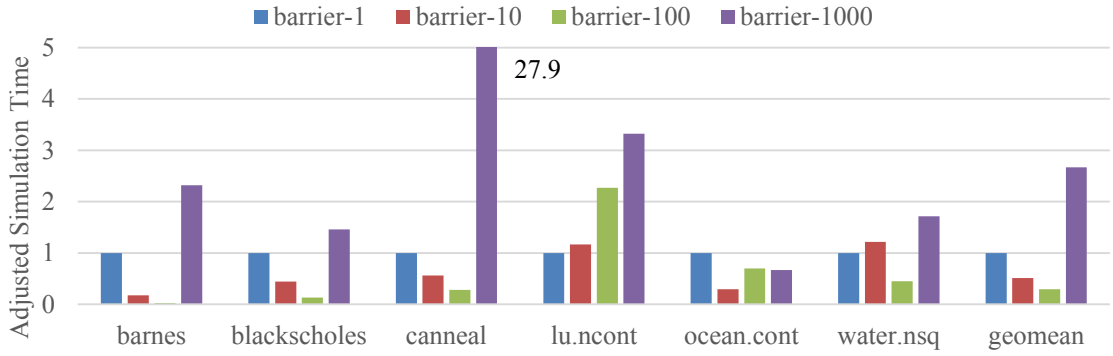


Figure 7: Simulation time for barrier synchronization at varying quantum sizes (see Fig. 13), multiplied by the number of runs needed for confidence derived in Table 1.

3.2 RANDOM PAIR USING WAIT-SIGNAL

Prior work introduced random-pair synchronization for parallel simulation of manycore architectures in the Graphite Simulator [48]. The Graphite simulator's random pair synchronization targets distributed host machines and relatively large synchronization quanta (1,000 to 100,000 cycles). When a thread attempts to synchronize with another thread, it uses the difference in core clocks and the simulation rate to estimate the amount of real time the slower thread would need to catch up. Based on the real-time estimate, the synchronizing thread makes a *sleep* system call to allow the target thread to catch up. On a CMP host, smaller communication latencies allow smaller synchronization quanta (around 100 cycles). With more frequent synchronization, using a *sleep* system call with an estimated time is too imprecise to compete with barrier synchronization.

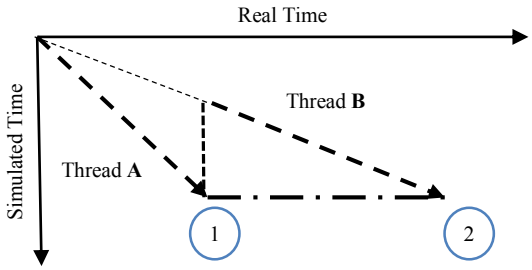


Figure 8: Example of random pair using wait-signal.

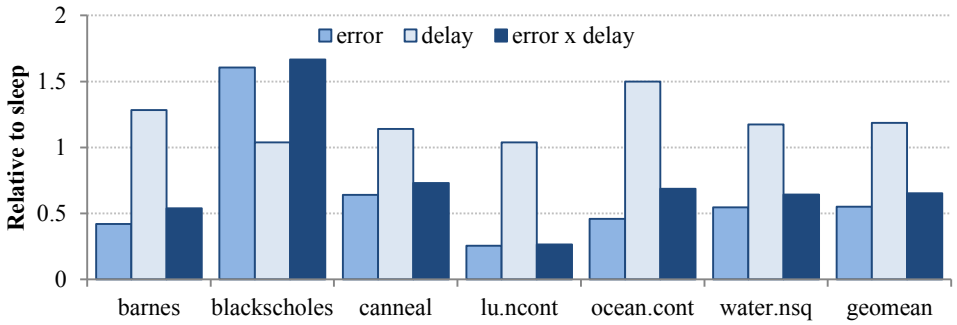


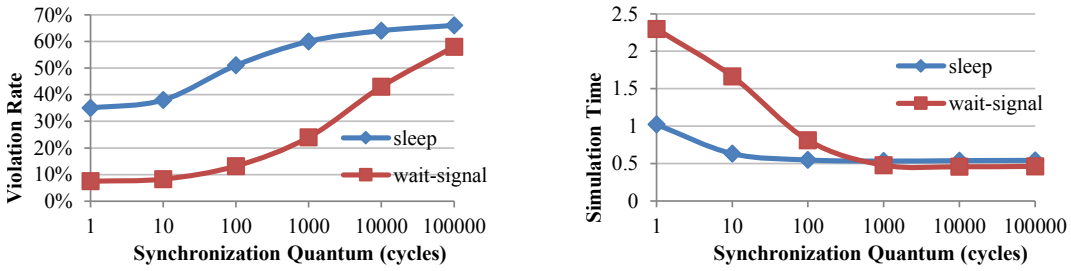
Figure 9: Error, delay, and error×delay for wait-signal synchronization normalized to sleep synchronization. Synchronization interval is 100-cycles.

As part of this work, random pair synchronization is adapted for CMP hosts via a wait-signal approach. Synchronizing simulation threads wait for a precise signal rather than using a sleep call. Each thread keeps a signal list, which tracks scheduled signals. Instead of sleeping, a thread schedules a remote signal and waits on a condition variable to resume. The target thread uses its signal list to signal the condition variable for a precise wake-up. Fig. 8 illustrates wait-signal random pair synchronization. When thread $\boldsymbol{A}$ reaches a synchronization point at real time point **1**, thread $\boldsymbol{A}$ discovers it is ahead of thread $\boldsymbol{B}$. Thread $\boldsymbol{A}$ places an entry onto $\boldsymbol{B}$'s signal list and begins waiting. Thread $\boldsymbol{B}$'s simulated time catches up to $\boldsymbol{A}$ at real time point **2**; thread $\boldsymbol{B}$ then signals thread $\boldsymbol{A}$ to resume simulation.

A comparison between sleep and wait-signal implementations for random-pair synchronization was performed for in the multicore simulation setting; evaluation details are discussed in Sec 3.3.1. The change in simulation fidelity and performance by using wait-signal rather than sleep is illustrated in Fig. 9. For most benchmarks[1], error (represented as the violation rate) is reduced significantly—by 45% on average with a 100-cycle quantum. Simulation time increases moderately (18.5% on average), with overall error×delay reduced by 35%.

The trends for error (violation rate) and delay (simulation time) with varying quantum size for the lu benchmark are shown in Figures 10a and 10b. Fig. 10a shows the violation rate with synchronization intervals ranging from 1 to 100,000 cycles; Fig. 10b shows the same trend for simulation time. At larger synchronization intervals, wait-signal and sleep behave

[1]Benchmarks are listed in the Appendix, see Table 7



(a) Violation Rate (b) Simulation Time

Figure 10: Error and delay comparisons between random-pair using sleep versus wait-signal with varying quantum size for the lu benchmark.

similarly; but with smaller synchronization intervals wait-signal pulls away from sleep in accuracy. This result is to be expected, as the sleep-based implementation targets distributed hosts using larger synchronization intervals and can adjust thread progress when synchronization is infrequent. At smaller quanta, the accuracy for a sleep-based implementation is limited due to the imprecise nature of sleeping. The more precise wait-signal random pair synchronization, however, achieves a much lower error rate, especially at 1000-cycle and shorter synchronization intervals. Below 100 cycles, wait-signal also takes a performance hit, as the sleep-based implementation allows threads to quickly complete while other threads are sleeping. In this work, wait-signal random-pair synchronization using an N-cycle quantum is referred to as *random-N*. From this point onward, references to random-pair are assumed to be the CMP-optimized (wait-signal) random-pair approach.

3.3 USING THE VIOLATION RATE TO MEASURE ERROR IN MULTICORE SIMULATION

In this section, both barrier and random-pair synchronization schemes are evaluated in the context of parallel multicore simulation. The advantages of using the violation rate as an error metric are experimentally shown. First, the correlation between the violation rate and various architectural metrics is shown to demonstrate that violations serve as a good proxy for error. Then, this section analyzes inaccuracies with architectural metric error that

arise as a result of the averaging effect. Architectural metric error, specifically cycles-per-instruction (CPI), is shown to have an unstable trend with varying synchronization interval sizes.

3.3.1 Evaluation

The Sniper multicore simulator was used as a baseline; it is described in Section 2.4.2. By default, the Sniper simulator attempts to synchronize at a trace granularity; a trace, in the PIN API, is a block of instructions with a single entrance and multiple exits. This limits the minimum synchronization quantum to the cycles needed to execute all instructions in a trace. To better study synchronization, the synchronization call was shifted inside the timing model so the simulator checks synchronization at the instruction granularity. With this change, all but a handful[2] of synchronization violations were removed when running with *barrier*-1. After these modifications, *barrier*-1 serves as the gold standard when measuring architectural metric error.

For violation tracking in CMP simulation, an added timestamp stores the thread index and timestamp for the latest access to each cache set. A violation occurs when an access timestamp is earlier than the stored value; the violating thread is the stored thread. Cache sets serve as the hardware unit rather than cache blocks because even if two accesses went to different blocks in the same set, changing the access order would affect the replacement policy.

Host and target machine specifications for this study are listed in Table 2. Benchmarks were selected from the Splash-2 and PARSEC benchmark suites; details can be found in Table 7. Three runs were executed per synchronization configuration, with the average presented; individual runs were used for the correlation study. Benchmarks were run to completion; simulation time was only measured during the region of interest, during which the detailed timing models were enabled.

[2]Remaining violations that occur with 1-cycle barrier synchronization are due to simulator artifacts, such as system call handling

Table 2: Machine Setup - Multicore Violation Study

Host Machine	Dual Socket Xeon X5680 @ 3.3GHz (2x 6 cores w/ HyperThreading). 96GB RAM
Target CPU	128 in-order cores, in-order @ 1GHz
Target Memory	L1 I -cache 16KB, 64B block, 2-way LRU L1 D-cache 16KB, 64B block, 2-way LRU Private L2 512KB, 64B block, 8-way LRU
Target Network	Mesh, 3-cycle/hop, no contention
Benchmarks (input)	SPLASH-2 (small) PARSEC (simmedium)

3.3.2 Synchronization Granularity Trends

The error trend for barrier and random-pair synchronization at varying quantum sizes is measured using the remote violation rate (shown in Fig 11) and CPI deviation from the *barrier*-1 baseline (shown in Fig. 12) to represent error. The trend for delay with varying synchronization intervals is shown in Fig. 13. While random-pair synchronization is faster than barrier synchronization, it trades off the added performance for reduced simulation fidelity. With shorter synchronization intervals (*random*-1 and *random*-10) random-pair synchronization exhibits minimal improvements in fidelity, despite incurring larger synchronization overheads. On average, random-pair synchronization has higher error than *barrier*-100 at all synchronization intervals. Thus, a distributed synchronization scheme would need to improve upon the accuracy of random-pair in order to compete with barrier synchronization.

Overall, synchronization violations and CPI error follow a similar trend, although the ratio of violations to CPI error varies based on the benchmark. In order to quantify the overall relation between the violation rate and CPI error, a study is performed correlating remote violations with CPI error. In addition to CPI, which is a straightforward metric, there are other architectural metrics of interest. Both CPI error and the violation rate are also correlated with error for L1 miss rate, L2 miss rate, and DRAM access count.

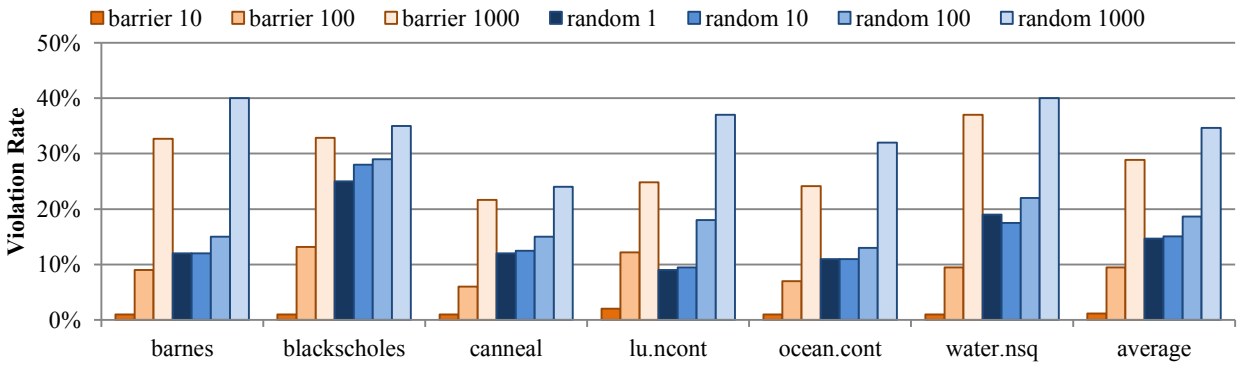


Figure 11: Remote violation rate for barrier and random-pair synchronization at varying quantum sizes.

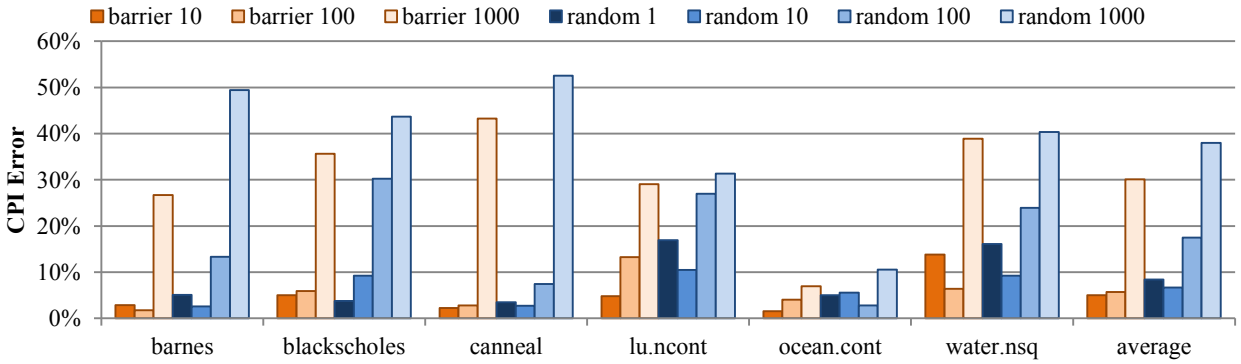


Figure 12: CPI deviation from *barrier*-1 baseline for barrier and random-pair synchronization at varying quantum sizes.

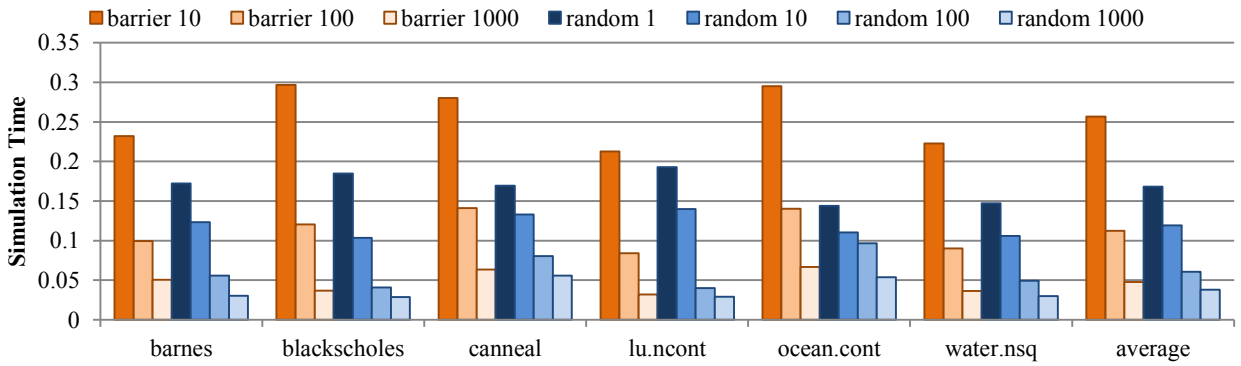


Figure 13: Simulation time, normalized to *barrier*-1 baseline, for barrier and random-pair synchronization at varying quantum sizes.

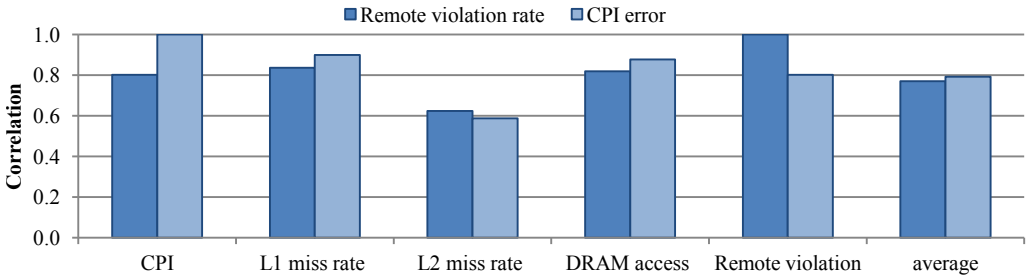


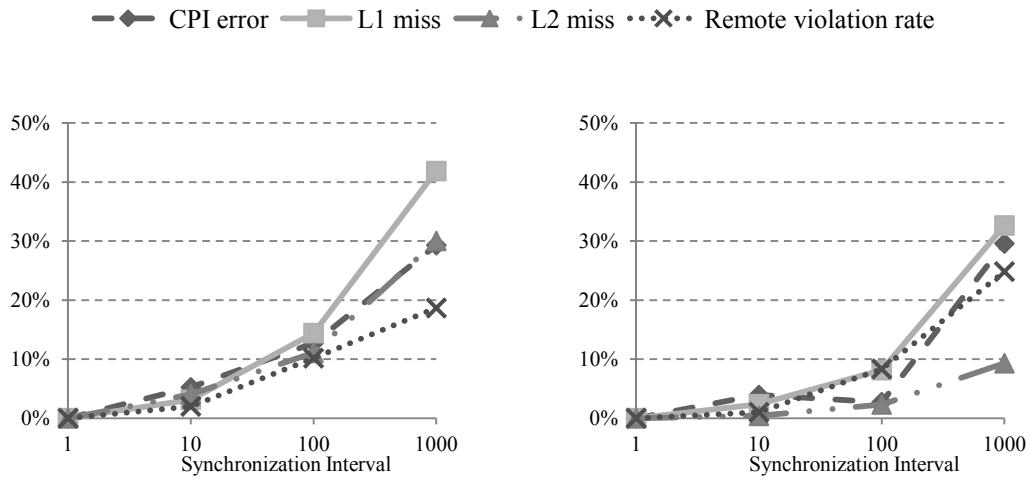
Figure 14: Correlation of remote violation rate and CPI error against error measured for other architectural metrics (shown on x-axis). The average omits self-correlation, i.e. the correlation between CPI and CPI which is 1.

3.3.3 Correlation Study

Figure 14 shows the results of the correlation study. Each bar represents the correlation between either CPI error or the remote violation rate and another error metric, as labeled on the x-axis. Sample points consist of all benchmarks and all quantum sizes, with 3 runs per configuration (see Sec. 3.3.1). As can be expected, both the remote violation rate and CPI error are correlated with the error rates for other measured architectural metrics. On average, correlation between remote violations and error for other metrics is only slightly lower than the correlation using CPI error instead. Using remote violations to predict error for other architectural metrics is just as accurate as using CPI error.

3.3.4 Architectural Metric Error

Lastly, architectural metric error can be subject to anomalous results where looser synchronization unintuitively reduces CPI error. Figure 15 shows CPI error with varying synchronization intervals for two benchmarks, lu and barnes. In Figure 15a, the error trend for lu scales as expected—less frequent synchronization results in higher error for all metrics; however, in Figure 15b the barnes benchmark shows an anomalous dip in CPI error with looser synchronization and more synchronization violations. This occurs because intervals where CPI is too high or low relative to the baseline may average out. The averaging effect for architectural metric error is further examined in Figure 16, which plots CPI over time for barnes. Because of intervals where CPI is unstable, comparing the average CPI



(a) lu (b) barnes

Figure 15: Error measured for various architectural metrics at varying barrier synchronization intervals.

of the two runs does not fully capture inaccuracies introduced from synchronization error. Synchronization violations have a baseline of zero violations and thus are not averaged out.

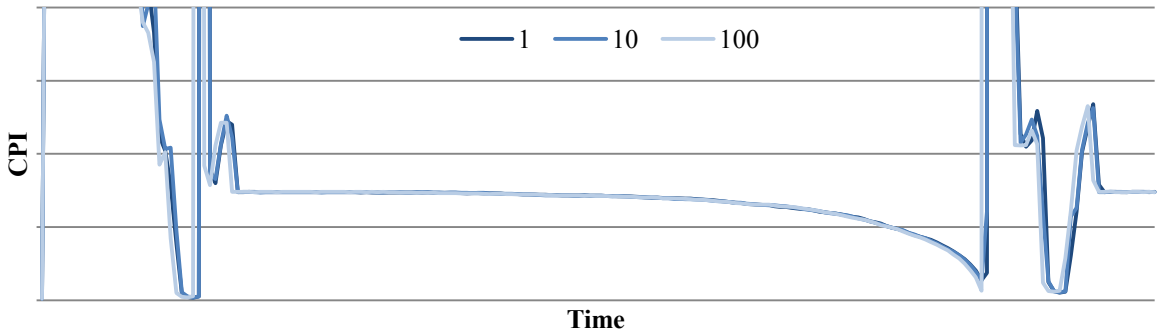


Figure 16: CPI over time for barnes with barrier synchronization at 1, 10, and 100 cycle intervals.

3.4 USING THE VIOLATION RATE TO MEASURE ERROR IN NETWORK-ON-CHIP SIMULATION

This section evaluates barrier and random-pair synchronization schemes in the context of parallel NoC simulation, using the error metrics of violation rate and flit latency error. The error and delay trade-offs for both synchronization schemes are demonstrated at varying synchronization intervals. This section shows both the violation rate and the flit latency error follow similar trends. Random-pair synchronization's performance is shown to be independent of the synchronization quantum, although it still has an impact on accuracy.

3.4.1 Setup

Simulation evaluation used the Hornet network-on-chip simulator as a baseline, which is described in Section 2.4.3. Because Hornet simulates each cycle in two clock phases, cycle-accurate simulation—with no violations—requires a barrier after both the positive and negative clock edges, which is denoted as *barrier*-0.5. With loose synchronization, a synchronization call is only made after the negative clock edge every N cycles. For random-pair synchronization, a synchronization call can still be made after both the positive and negative clock edges, which is denoted as *random*-0.5; unlike *barrier*-0.5, *random*-0.5 is not cycle-accurate.

Violations are tracked in a similar manner to that described in Section 3.1 and Figure 5. To track violations for NoC simulation specifically, each flit tracks its injection time and age—these are summed to determine the flit's simulated timestamp. When a flit traverses the crossbar, its timestamp is compared against the latest timestamp recorded by the crossbar. If a flit traverses the crossbar with a lower timestamp, a violation is recorded.

Host machine and target network parameters are listed in Table 3. Synthetic traffic patterns were used to generate events. For the static traffic patterns (bitcomp, shuffle, transpose), each source tile sends packets to one destination tile whose index is determined by the pattern. For example, with the bitcomp pattern each tile determines packet destinations by inverting the bits corresponding to its tile index; further details on traffic patterns can be

Table 3: Machine Setup - NoC Violation Study

Host Machine	Dual Socket Xeon E5-2470 @ 2.3GHz (2x 8 cores w/ HyperThreading). 128GB RAM
Host Threads	32
Target Network	256 tiles
Network Layout	2-D mesh, 16 × 16
Network Routing	X-Y dimension-order
Network Routers	Mesh, 3-cycle/hop
Traffic	5-flit packets, injection period 100
Patterns	bitcomp, shuffle, transpose, random

found in the Appendix, Table 8. For random traffic, a remote destination is selected from all tiles uniformly at random for every generated packet. Each configuration was simulated three times, with one million cycles per run.

3.4.2 Synchronization Granularity Trends

Figures 17 and 18 show violation rate and latency error, respectively, for barrier and random-pair synchronization schemes at various synchronization quanta. *Barrier*-0.5, which synchronizes all threads after each positive and negative edge, serves as a baseline with zero violations and the highest delay. Using larger synchronization intervals significantly reduces accuracy for both barrier and random-pair synchronization. Latency error sees a large increase with the less accurate random-pair schemes, *random*-5 and *random*-10.

As with multicore simulation, the violation rate trend appears similar to the trend for packet latency error. The correlation between the violation rate and average flit latency error was measured over all runs to be 0.83. Other architectural metrics, such as link power or utilization, were based solely on the traffic pattern and were independent of the synchronization scheme.

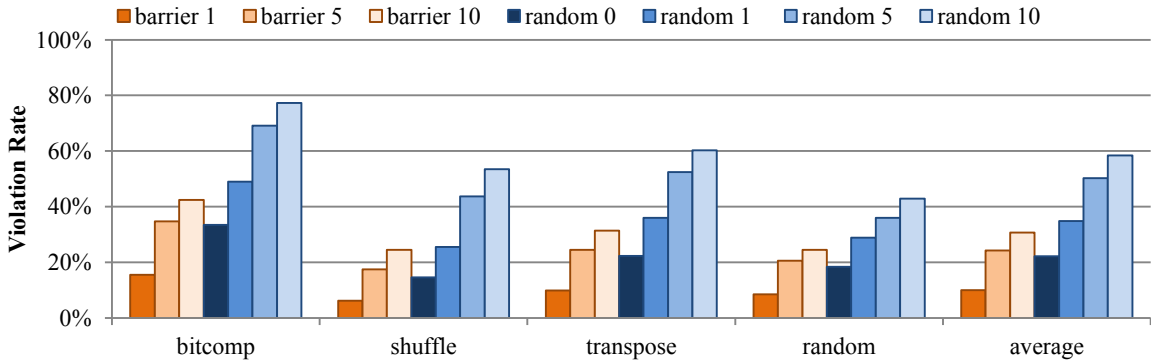


Figure 17: Violation rate for barrier and random schemes with 0.5, 1, 5, and 10 cycle synchronization quantum.

Performance for barrier and random-pair synchronization are shown in Figure 19. Barrier synchronization steadily speeds up with looser synchronization. For random-pair synchronization, the synchronization quantum does not impact performance heavily; thus it is best to use a 0.5-cycle quantum (two synchronizations per cycle). The most accurate random-pair synchronization scheme, *random*-0.5 has a slightly better performance-accuracy trade-off compared to *barrier*-10, the barrier scheme with the closest simulation time. *Random*-0.5 has a 27.6% lower violation rate and a 25.1% lower latency error; meanwhile, it takes 6.1% longer to complete simulation.

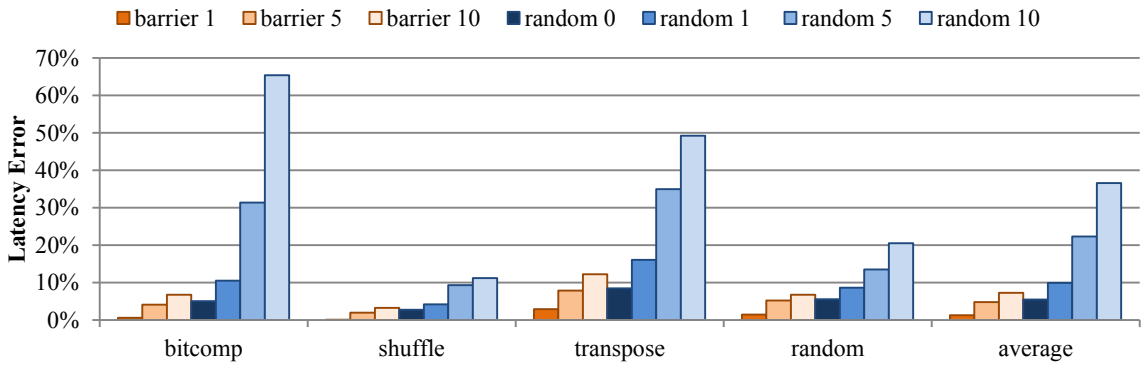


Figure 18: Latency error for barrier and random schemes with 0.5, 1, 5, and 10 cycle synchronization quantum.

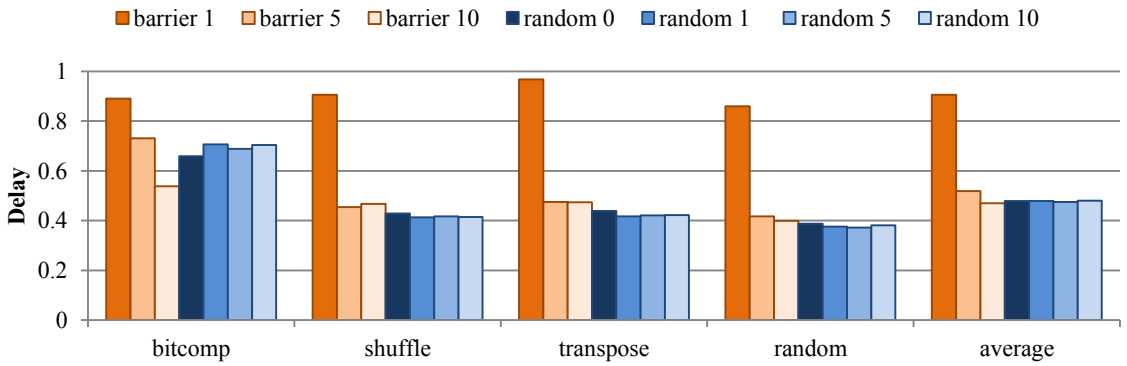


Figure 19: Delay for barrier and random schemes with 0.5, 1, 5, and 10 cycle synchronization quantum. Values normalized to *barrier*-0.5.

3.5 CONCLUSION

As more researchers turn to parallel simulation for performance modeling, it becomes more important to understand and be able to quantify synchronization error. This chapter profiled two existing synchronization schemes, barrier and random-pair, in two separate parallel simulation settings: multicore simulation and network-on-chip simulation. As part of the analysis of random-pair synchronization, a wait-signal implementation was introduced for accurate synchronization on CMP hosts. In addition, this chapter showed that the synchronization violation rate is a good metric for synchronization error. The violation rate is more easily measurable than architectural metric error in that it does not require a baseline. Moreover, the violation rate is a good indicator for predicting various architectural metric error, such as CPI or network latency error, and is not subject to an averaging effect.

4.0 WEIGHTED-TUPLE SYNCHRONIZATION

This chapter describes a new synchronization policy, weighted-tuple synchronization, which is a distributed synchronization policy similar to random-pair synchronization. It makes two key generalizations over random-pair synchronization that lead to improved accuracy: tuple synchronization and weighted target selection. Tuple synchronization involves selecting multiple synchronization targets at a time. It can be viewed as a generalization of synchronization granularity between random-pair synchronization (where one target is selected) and barrier synchronization (where all active threads are selected for synchronization). For weighted target selection, a synchronizing thread selects synchronization targets using a weighted distribution—the weight is a heuristic such as the number of triggered violations or the relative difference in simulation times. A portion this chapter was published in [51].

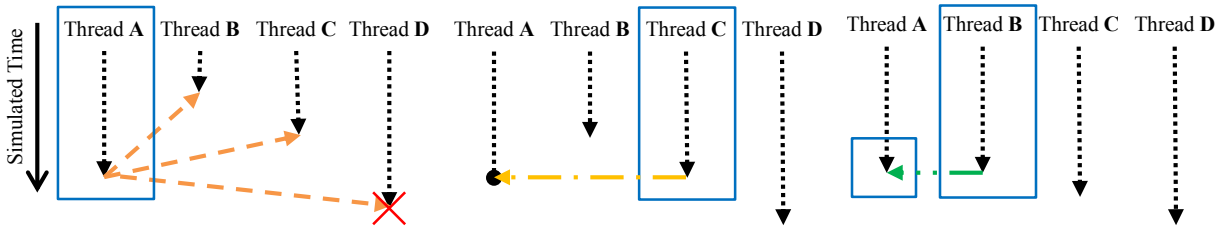
After weighted-tuple synchronization is described, it is evaluated in two simulator settings. The first simulator setting is parallel multicore simulation. In this setting, the default synchronization quantum is 100 cycles [9]; quanta of ten and one thousand cycles are also considered. In addition, remote cache accesses are performed directly via a shared memory access, allowing a thread to access another thread's state. Thus, a violation only occurs when a thread makes a remote access with an incorrect timestamp. The second simulator setting is a parallel network-on-chip simulator. For accurate network simulation, synchronization occurs between twice a cycle and once every ten cycles [43]. In addition, flits are passed off between threads as they travel from source to destination; if any of the threads which handle a flit are unsynchronized, the flit can cause a violation. In this setting, when a flit arrives at the wrong time, multiple threads are potentially responsible.

4.1 TUPLE SYNCHRONIZATION

Using the multicore simulation setting, an examination was performed on the number of synchronization "waits" executed (i.e., a thread pausing simulation to wait for another thread); details on the experiment can be found in Section 3.3. The examination revealed that the synchronization waits were roughly the same for *random*-1, *random*-10, and *random*-100. With more frequent synchronization (a smaller quantum), there is a lower probability of randomly selecting a target with a significantly lower clock. Thus, many synchronization attempts fail to synchronize with threads that are far behind.

To improve the probability that a synchronization attempt results in the thread actually waiting for a slower thread, this dissertation introduces a generalized distributed synchronization scheme, tuple synchronization. While random-pair synchronization selects one synchronization target, under tuple synchronization multiple targets are selected to form a *synchronization tuple*. This generalized synchronization scheme encompasses a range of prior synchronization schemes. At one extreme lies random-pair synchronization, where a thread selects one synchronization target—resulting in high performance but low accuracy. At the other extreme lies barrier synchronization, which can be thought of as a case where a thread selects all other threads as synchronization targets; barrier synchronization maximizes accuracy at the cost of simulation speed. Tuple synchronization generalizes the number of synchronization targets, allowing for a middle ground which is both accurate and has low overhead.

Random tuple synchronization, using the wait-signal implementation described in Section 3.2, is illustrated in Figure 20 (an example with three targets). Instead of selecting one thread at random, thread $\boldsymbol{A}$ randomly selects three other threads: $\boldsymbol{B}$, $\boldsymbol{C}$ and $\boldsymbol{D}$. Thread $\boldsymbol{A}$ then compares its simulated time with each target thread. Having progressed further than threads $\boldsymbol{B}$ and $\boldsymbol{C}$, thread $\boldsymbol{A}$ schedules itself on the signal lists for both cores $\boldsymbol{B}$ and $\boldsymbol{C}$ (Figure 20a). Because thread $\boldsymbol{D}$ is ahead of $\boldsymbol{A}$, it is ignored by thread $\boldsymbol{A}$ during synchronization. Thread $\boldsymbol{A}$ also keeps a counter in shared memory for the number of outstanding signal list items before waiting (two in this case). When thread $\boldsymbol{C}$ catches up with thread $\boldsymbol{A}$, $\boldsymbol{C}$ decrements the outstanding signal counter (Figure 20b). When thread $\boldsymbol{B}$ catches up with thread



(a) Schedule Signals (b) Decrement Count (c) Signal

Figure 20: Three target tuple synchronization example using wait-signal.

A, it also decrements the outstanding-signal counter. As thread ***B*** decremented the counter to zero, it signals thread ***A***, allowing threads ***A*** to resume simulation (Figure 20c). Threads ***B*** and ***C*** continue their own simulation after signaling ***A***. Random tuple synchronization with an N-cycle quantum and K targets is denoted as *random-N T-K*.

4.2 WEIGHTED TARGET SELECTION

With random-pair, each thread selects its synchronization target uniformly at random. However, for a synchronizing thread, some synchronization candidate threads may be more appropriate than others: a candidate thread may cause more violations due to heavier communication or may be progressing slower and need time to catch up. This work proposes weighted target selection, which assigns synchronization candidate threads a synchronization weight according to some heuristic. The two heuristics studied here are the violations and simulated time skew.

4.2.1 Weighted Targets by Violations

To motivate violations as a weighted target heuristic, an analysis was performed on the distribution of violations in a multicore setting using random-pair synchronization (see Section 3.3 for simulation details and Appendix Table 7 for benchmark details). Triggered violations were tracked and stored into a shared memory structure. For each thread's violations, the threads which triggered its violations were sorted by triggered violations and the violations

Table 4: Violation Coverage from Top 13 Cores

Benchmark	Random-10	Random-100	Random-1000
barnes	89%	81%	75%
blackscholes	91&	91%	77%
canneal	78%	76%	46%
lu.ncont	91%	84%	70%
ocean.cont	54%	54%	40%
water.nsq	65%	67%	69%
average	77%	75%	61%

from the top 13 threads (to represent 10% of the 128 total threads) were tallied—this sum was then averaged across all threads. Table 4 shows the total percentage of violations covered by the threads which triggered the most violations. The analysis shows a large percentage of violations are triggered by a relatively small number of threads.

This study motivates the use of a weighted-violation synchronization approach. Instead of selecting a target uniformly at random, synchronization targets are selected with weight equal to the number of violations triggered by the thread. If thread ***A*** triggered 30% of core ***B***'s violations, when ***B*** selects a target it has a 30% chance of picking thread ***A***. The weighted-violation scheme with a quantum of N cycles is denoted as *weighted-vio-N*.

4.2.2 Weighted Targets by Clock Skew

Depending on the simulation setting, violations may not be concentrated into a small number of threads. In addition, for some simulations messages interact with more than two threads. For example, in network-on-chip simulation, a flit passes through multiple tiles between the source and destination. The flit's arrival time at the destination is thus determined by multiple threads. If the flit does cause a violation, it is still unclear which threads should be synchronized.

The second weighted heuristic this work investigates is synchronizing with weight equal to a thread's clock skew relative to the synchronizing thread—the synchronizing thread compares its simulated time with synchronization candidate's simulated time (which is accessed through shared memory). For example, a thread is twice as likely to synchronize with a thread which is 200 cycles behind than a thread which is 100 cycles behind. Because threads synchronize more often with the slowest threads, weighing targets based on the simulated time differential will reduce the overall clock skew in the system; however, the random element ensures synchronizing threads do not all select and wait for the same target thread. This weighted scheme is denoted as *weighted-time-N*.

4.2.3 Combining Tuple Synchronization with Weighted Target Selection

Weighted-tuple synchronization is derived by combining weighted target selection with tuple synchronization. At each synchronization period, each thread forms a synchronization tuple by selecting one or more synchronization targets. The synchronizing threads' targets are selected with varying weight—either based on their violation trigger rate or their current time difference. A weighted-tuple scheme using a quantum size of N which selects K targets per synchronization attempt is denoted as *weighted-vio-N T-K* when weighing by violations and as *weighted-time-N T-K* when weighing by clock skew.

4.3 WEIGHTED-TUPLE FOR MULTICORE SIMULATION

The first study analyzing weighted-tuple synchronization targets parallel multicore simulation to analyze the error delay trade-off. Unless otherwise stated, error is measured as the remote violation rate, and delay is the simulation time for the region of interest. A combined metric, error×delay, is used to directly compare different synchronization schemes in order to represent the trade-off between error and simulation time. The multiplicative error×delay metric is not well defined when error approaches zero, but does allow comparisons between synchronization schemes with non-zero error (e.g., when the quantum size is at least 100

cycles). While error×delay does not do a good job of expressing behavior for *barrier*-1 and *barrier*-10, these schemes have prohibitively high synchronization costs and are unlikely to be of interest to researchers using parallel simulators for multicore architecture research. Error, delay, and error×delay are evaluated for synchronization quanta of 10, 100 and 1000 cycles. The synchronization schemes evaluated are *barrier*, *random-tuple*, and *weighted-tuple* synchronization.

Violations are tracked during simulation both to measure error and for weighted-violation target selection. Each thread maintains a structure in shared memory storing its violations and the distribution of threads triggering violations; this is updated by other threads as they trigger violations. Because threads host a single core and directly access cache state for their core, the violation trigger distribution used for weighted-violation target selection is fairly representative of a core's access traffic.

Full-system multicore simulators must model system and synchronization calls; these calls can cause the thread's functional progress to become blocked (such as waiting for a barrier executed by the simulated program). When a thread's functional simulation becomes blocked by a system call, its timing progress is also blocked. To avoid deadlock, a blocked simulation thread immediately notifies all threads waiting for it to resume simulation (or decreases the signal counter for tuple synchronization). Threads blocked by simulated system calls are never the target for synchronization, and are additionally excluded for weighted-time synchronization as potential targets as a blocked thread's simulated time tends to lag far behind until it becomes unblocked.

4.3.1 Evaluation

This evaluation used the Sniper multicore simulator described in Section 2.4.2, which was modified for precise synchronization and violation tracking as described in Section 3.3.1. The simulation was set up in the same manner as Section 3.3; host and target machine parameters can be found in Table 2. The same benchmarks used in Section 3.3 were used here—benchmark details can be found in Appendix A, Table 7. Unless otherwise stated, error denotes the remote violation rate. Because *random*-1 was shown to have nearly-identical error

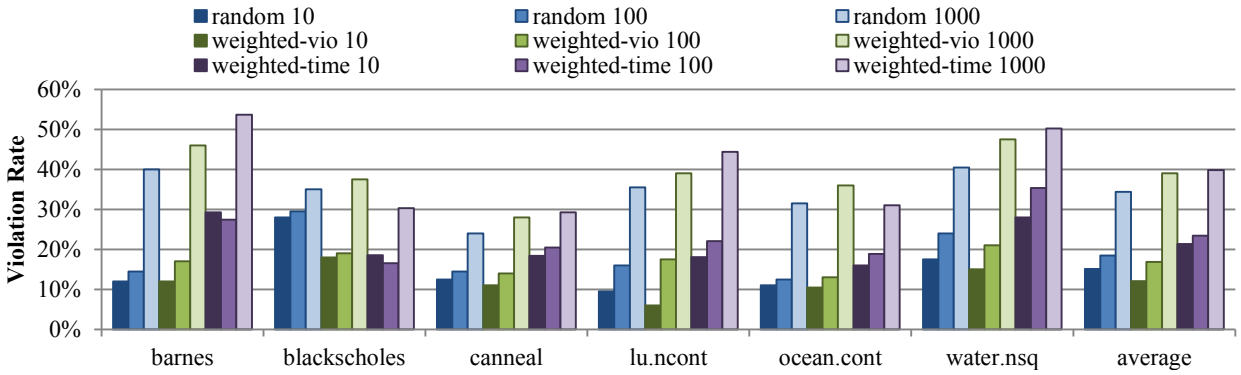


Figure 21: Error for random, weighted-vio, and weighted-time target selection with 1 synchronization target for 10, 100, and 1000 cycle synchronization quanta.

as *random*-10 in Section 3.3, results for a quantum size of 1 are omitted for the multicore simulation setting.

4.3.2 Weighted Targets

A comparison between uniform random, weighted-vio and weighted-time target selection schemes is shown in Figure 21. Weighted-time shows poor accuracy compared to weighted-violation target selection in this simulation setting, especially at smaller synchronization intervals. As core-to-core communication is very localized, weighted-time synchronization wastes synchronization opportunities on cores which are unlikely to cause violations. The remaining results for multicore simulation omit weighted-time synchronization and focus on weighted-vio target selection.

Figures 22, 23, and 24 illustrate error (violation rate), delay (simulation time), and error×delay for quantum sizes of 10, 100, and 1000, respectively. Error× delay is normalized to barrier synchronization. Random tuple and weighted-violation tuple synchronization schemes are evaluated in addition to the barrier baseline.

Accuracy and performance by using weighted-vio random synchronization without tuple synchronization can be seen by examining the *weighted-vio T1* bars. Compared to random-pair, weighted-vio-T1 synchronization has an average 20% reduction in violations for 10-

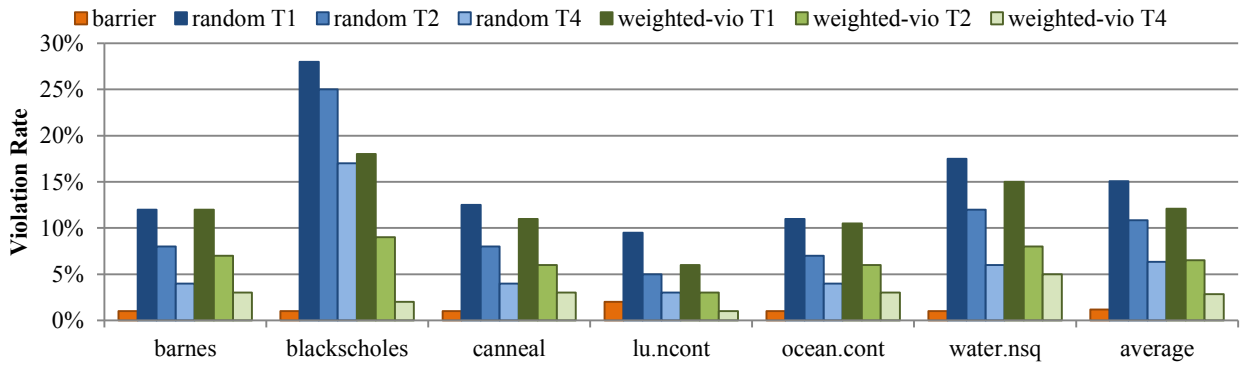
cycle synchronization, a 9% reduction for 100-cycle synchronization, and a 13% increase in violations with 1000-cycle synchronization. From Table 4 on page 43, there is worse coverage for the top violation-triggering cores at larger quantum sizes. With infrequent synchronization, cores drift apart more, making any synchronization target a candidate for a significant number of violations and as such, useful for reducing error rate. Thus, keeping an even distribution of synchronization targets by selecting targets uniformly at random does a better job of reducing violations for large quantum sizes.

4.3.3 Tuple Synchronization

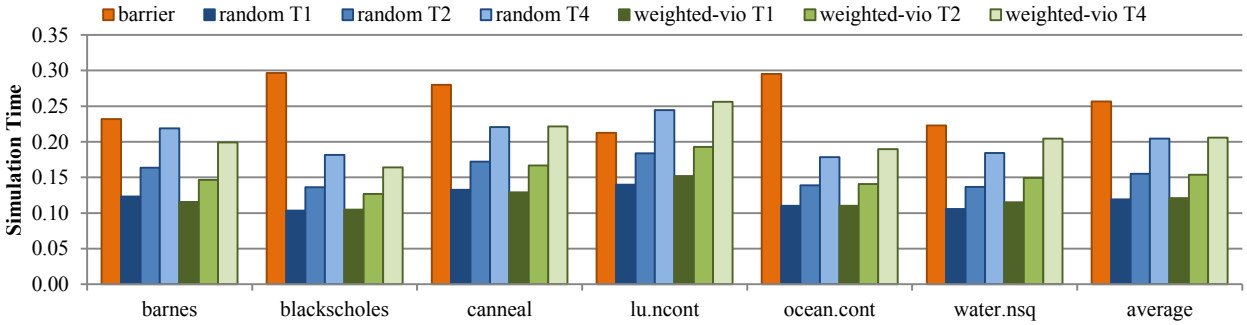
Synchronizing with 2 or 4 targets improves error at all synchronization quanta, as shown by the *random T*2 and *random T*4 bars. The fidelity improvement is especially noticeable for the shorter 10-cycle quantum, where random-pair has poor accuracy. With random target selection, 2 and 4 targets reduce the violation rate by 28% and 58%, respectively, over random-pair. As expected, using more synchronization targets incurs additional overheads: random T2 and T4 are 30% and 72% slower than random-pair. Despite the performance loss, error×delay improvements from T2 and T4 tuple synchronization are 7% and 30%, respectively.

While smaller improvements are observed at 100-cycle and 1000-cycle quanta, tuple synchronization also incurs lower overhead and overall is still beneficial; average error×delay improvements from random-pair to random T4 are 25% and 5% for 100 and 1000 cycle quanta, respectively. Because random-pair does a better job of selecting targets with larger quanta, adding additional synchronization targets results in a smaller accuracy improvement. For example, *random*-1000 *T*4 has 18% fewer average violations than *random*-1000.

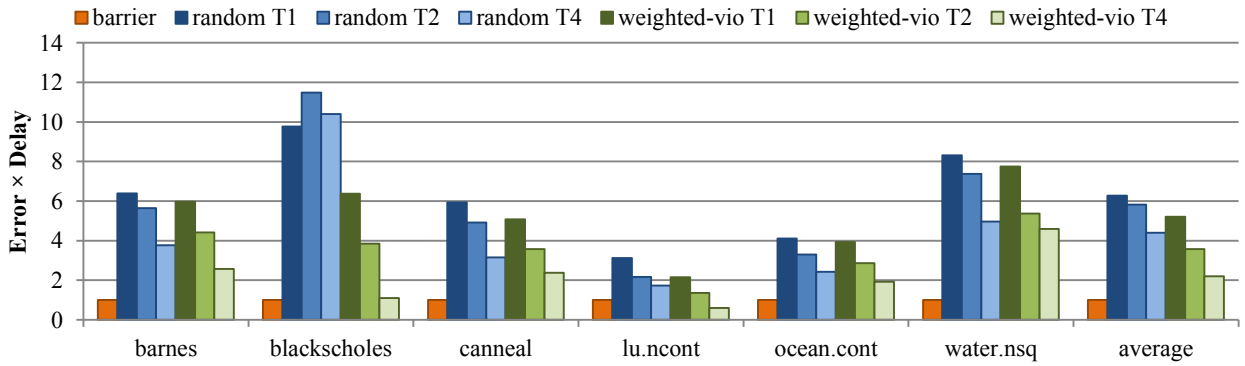
While tuple synchronization generally improves accuracy over random-pair, it also retains improved performance compared to barrier synchronization for a beneficial middle ground. Random T4 has 20%, 10%, and 8% shorter average simulation times compared to barrier with 10, 100, and 1000-cycle synchronization, respectively.



(a) Error (violation rate)

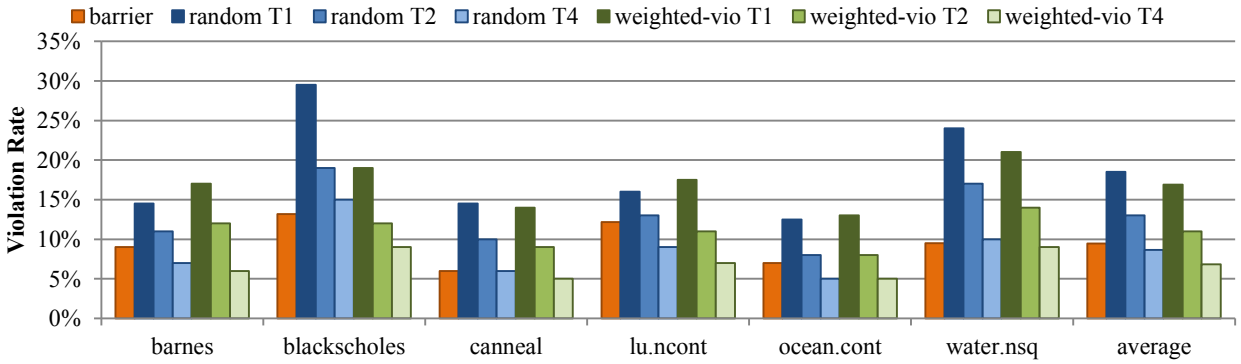


(b) Delay (simulation time), normalized to *barrier*-1

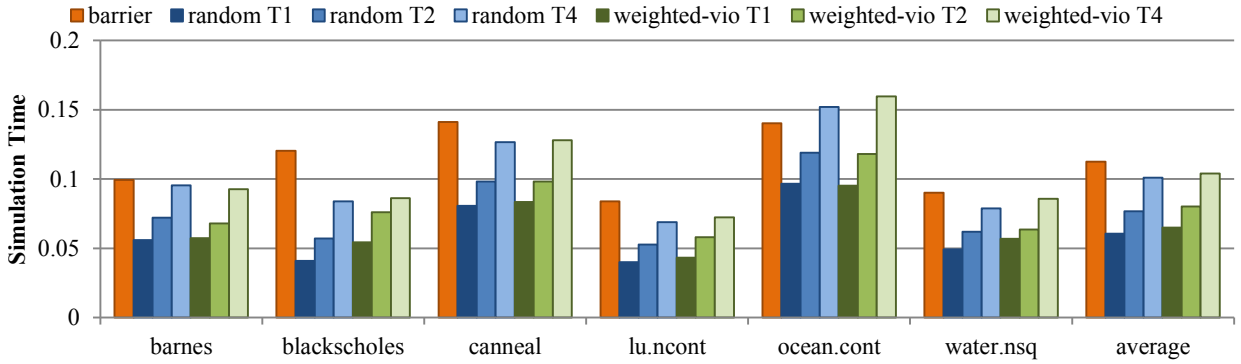


(c) Error×Delay, normalized to *barrier*-10

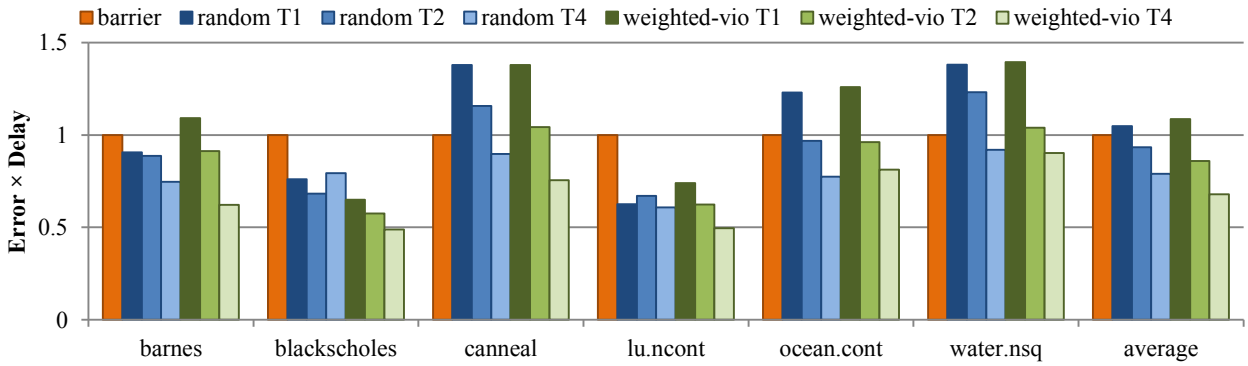
Figure 22: Error, delay, and error×delay for random and weighted-vio schemes with 1, 2, and 4 synchronization targets for a 10-cycle quantum.



(a) Error (violation rate)

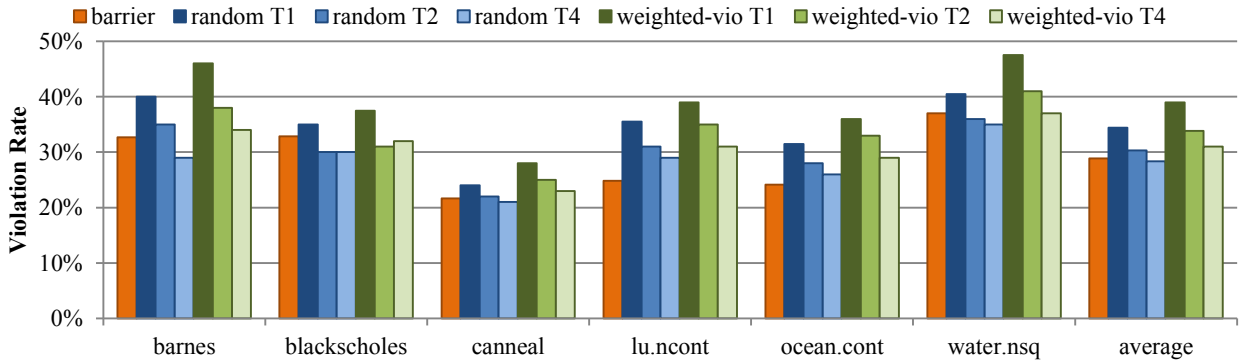


(b) Delay (simulation time), normalized to *barrier*-1

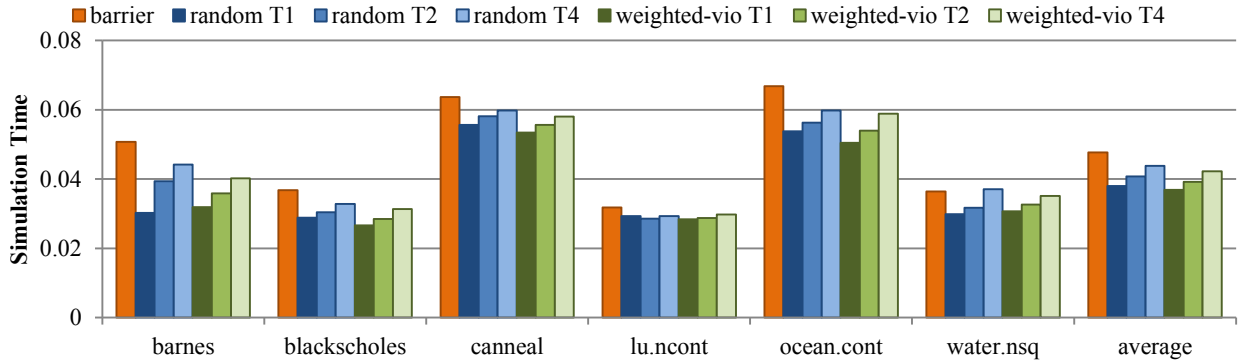


(c) Error×Delay, normalized to *barrier*-100

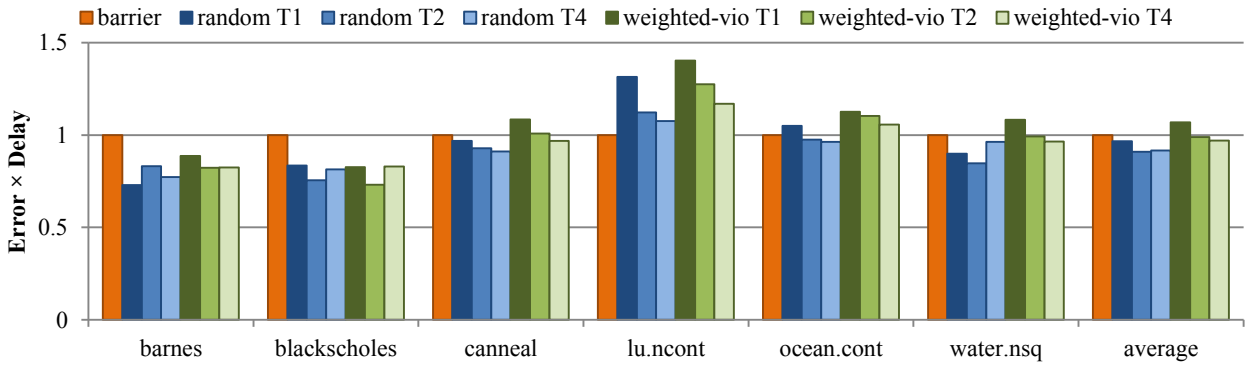
Figure 23: Error, delay, and error×delay for random and weighted-vio schemes with 1, 2, and 4 synchronization targets for a 100-cycle quantum.



(a) Error (violation rate)



(b) Delay (simulation time), normalized to *barrier*-1



(c) Error×Delay, normalized to *barrier*-1000

Figure 24: Error, delay, and error×delay for random and weighted-vio schemes with 1, 2, and 4 synchronization targets for a 1000-cycle quantum.

4.3.4 Weighted-Tuple Synchronization

Combining the use of synchronization tuples with weighted-vio target selection further improves simulation fidelity for smaller quantum sizes, illustrated by the *weighted-vio TK* bars. Weighted-vio-T4 has 65% and 35% lower error×delay versus random-pair with 10 and 100 cycle quanta respectively, and an 0.3% higher error×delay for the 1000-cycle quantum.

A surprising result is that *weighted*-100 *T*4 has lower error than *barrier*-100 for all benchmarks, with a 28% reduction in the violation rate. This is due to the activity surge when a barrier is released. With a large number of target cores, native simulation results in many more simulation threads than host cores. As the last thread reaches the barrier, all threads become active and contend for host cores. As a limited number of simulation threads are allowed to execute at a time, the core clocks will drift apart. With a decentralized synchronization scheme like weighted-tuple, threads wait independently of each other. In contrast to barrier, this keeps simulation progress steady. Moreover, the threads which do drift apart and are not synchronized are unlikely to cause violations with one another because of the violation-weighted target selection.

The average error and delay trends for weighted-tuple with varying quantum size are shown in Figure 25—values are normalized to *barrier*-100. The accuracy for the *weighted-vio T4* scheme scales down with shorter synchronization intervals, especially when compared to the error trend for *random T1*. Delay does decrease significantly moving from 10-cycle to 100-cycle synchronization with 4 synchronization targets, which offsets the reduction in fidelity for a roughly equivalent error×delay. Relative to the *barrier*-100 baseline, weighted-vio T4 synchronization has 42% and 41% reductions with 10 and 100 cycle synchronization intervals, respectively; *weighted-vio*-1000 *T*4 has 10% higher error×delay.

4.3.5 CPI Error

In Fig. 26 CPI error is substituted for remote violation rate as the error metric for a 100-cycle synchronization quantum, with CPI error shown in Fig. 26a and CPI error×delay shown in Fig. 26b. Because CPI error is less predictable, the scheme with the lowest error×delay is *weighted-vio*-100 *T*2 with a 38.9% improvement over *random-pair* and a 6.8% improvement

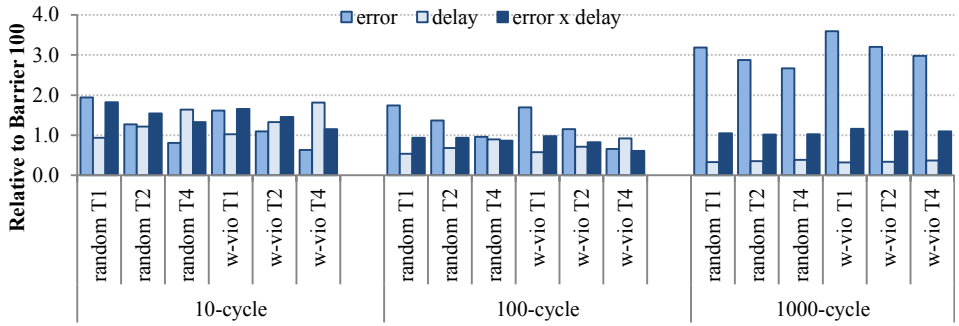
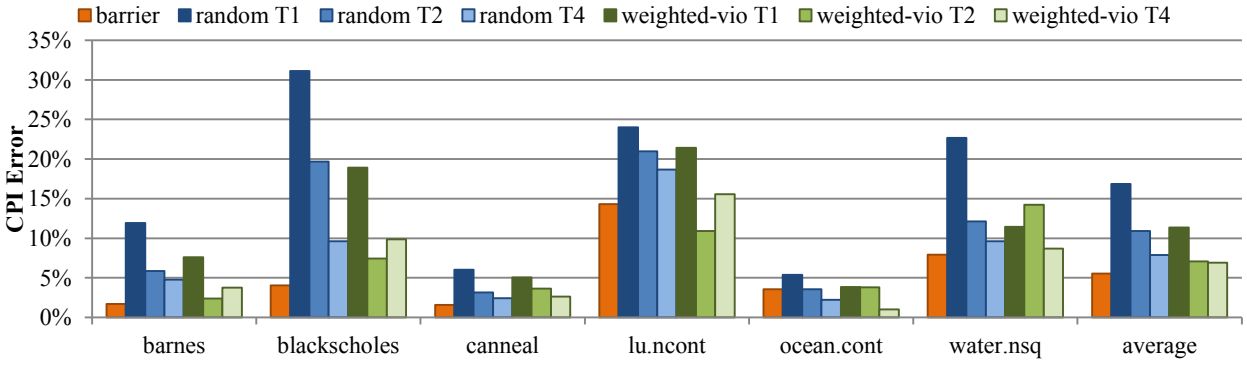
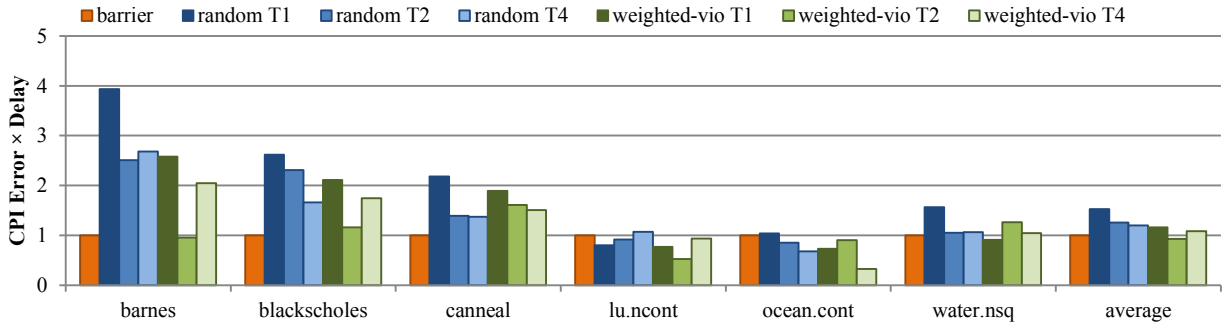


Figure 25: Error, delay, and error×delay for random and weighted-vio (w-vio) schemes with 1, 2, and 4 synchronization targets for 10, 100, and 1000 cycle synchronization intervals. Values normalized to *barrier*-100.

over *barrier*. As shown in Section 3.3, the relation between CPI error and the violation rate varies from benchmark to benchmark; however, the relative accuracy of synchronization schemes when using CPI error is similar to the trend when using the violation rate (Figure 23a).



(a) CPI Error



(b) CPI Error×Delay, normalized to *barrier*-100

Figure 26: Error and error×delay, where error is represented by CPI error, for random and weighted-vio schemes with 1, 2, and 4 synchronization targets for a 100-cycle quantum.

4.4 WEIGHTED-TUPLE FOR NETWORK-ON-CHIP SIMULATION

This section evaluates weighted-tuple synchronization for NoC simulation. The *barrier*-0.5 policy serves as the baseline, with no violations and the longest simulation time. The error metrics examined are violation rate and packet latency deviation from the baseline.

To track the violation trigger distribution for weighted target selection, each time a violation occurs the destination thread records the violating flit's source tile's thread. The distribution of source thread violations is used by weighted target selection to select synchronization targets. A key difference with the multicore simulator setting from Section 4.3 is that each thread in the Hornet NoC simulator simulates multiple tiles. Unlike multicore simulation, where the violation trigger distribution is representative of core-to-core traffic patterns, in NoC simulation the violation trigger distribution is an aggregation of violations for all tiles simulated by a thread. Furthermore, a flit can cause violations at any step along its path, not just its destination; thus there is a degree of topology-based locality added to the violation trigger distribution.

4.4.1 Evaluation

Weighted-tuple synchronization was evaluated using the Hornet NoC simulator. The simulation was set up in the same manner as Section 3.4; host and target machine parameters are listed in Table 3. As shown in Fig. 19, using a 0.5-cycle synchronization interval (synchronization after both positive and negative clock edges) had the same performance as a 1-cycle synchronization interval for distributed synchronization schemes (random-pair and weighted-tuple); moreover, accuracy was better for 0.5-cycle intervals. Thus, only 0.5-cycle interval results are shown for distributed synchronization schemes.

Uniform random, weighted-violation, and weighted-time target selection are evaluated with 1, 2, 4, and 8 targets. Error, represented as violation rate and latency error, is shown in Figures 27 and 28, respectively. Figure 29 illustrates delay. All distributed schemes run much faster than *barrier*-0.5 and *barrier*-1; simulation times are comparable to *barrier*-10.

4.4.2 Weighted Target Selection

Weighing target selection by violations does not reduce error compared to uniform random target selection for network-on-chip simulation. For network-on-chip simulation, each message passes through multiple tiles—a clock skew in any intermediate tile can cause a violation. Thus, minimizing clock skew is better for this type of simulation (where messages interact with more than two threads).

Using clock skew as the heuristic for weighted target selection reduces error significantly. For single-target synchronization, weighing target selection with clock skew reduces average error to 6.8% for the violation rate and 2.3% latency error on average. Compared to uniform random target selection, weighing by clock skew reduces error by 47.7%. The error reduction brings weighted-time-1t very close to *barrier*-1 even before adding tuple synchronization. The *weighted-time* synchronization scheme does run somewhat slower than the other distributed synchronization schemes, likely because the memory locations corresponding to each thread's synchronization time are more heavily contended for than those corresponding to violation rates.

4.4.3 Tuple Synchronization

Increasing the number of synchronization targets steadily improves accuracy for all target selection policies. Violations from 1T to 8T decrease by 57.2%, 47.5%, and 82.5% for random, weighted-vio, and weighted-time target selection, respectively. Simulation slows down somewhat with more targets, but is still much faster than barrier with 0.5 or 1-cycle synchronization. Overall, *random-8t* has 7.2% fewer violations and reduces simulation time by 51.0% compared to *barrier*-1.

4.4.4 Weighted-Tuple Synchronization

As in the multicore simulation setting, combining weighted target selection with tuple synchronization further improves accuracy. The 47.7% lower violation rate *weighted-time* has over *random* for one target increases to a 77.0% violation rate reduction for eight target

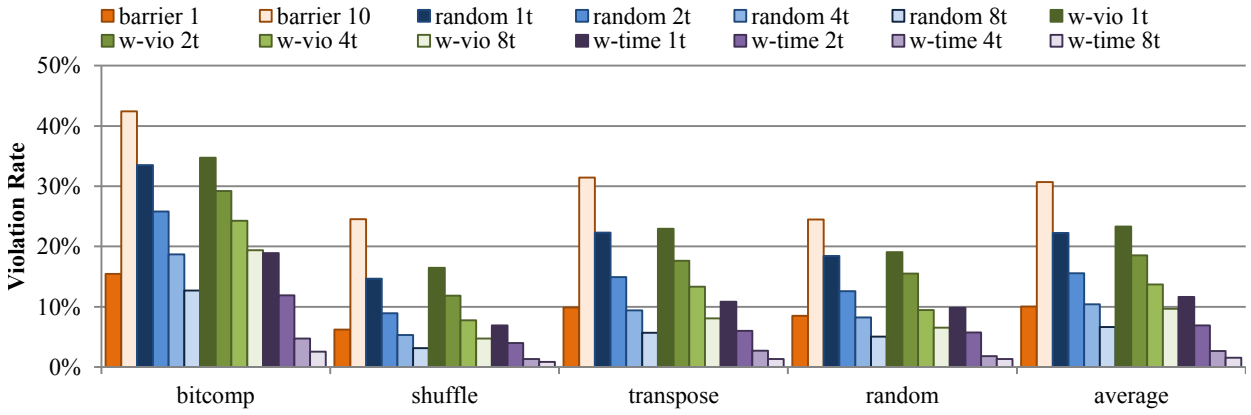


Figure 27: Violation rate for random and weighted schemes 1, 2, 4, and 8 synchronization targets, synchronizing twice every cycle. *Barrier*-1 and *barrier*-10 are included for reference.

synchronization. Weighted-time with two or more targets is both more accurate and faster than *barrier*-1 synchronization, reducing the violtion rate by 84.8% and simulation time by 40.8% with eight targets. Unless absolute accuracy is required (*barrier*-0.5), weighted-tuple synchronization is a superior form of loose synchronization in both accuracy and performance.

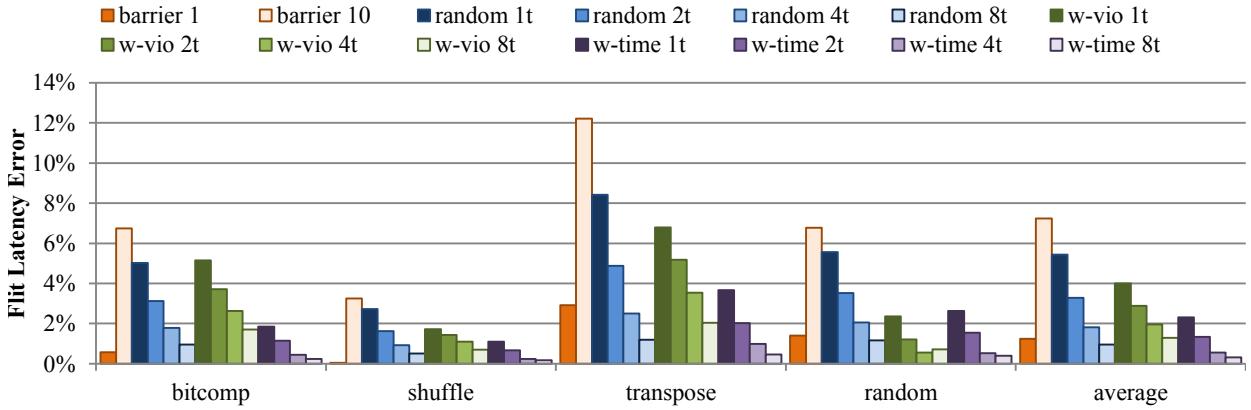


Figure 28: Latency error for random and weighted schemes 1, 2, 4, and 8 synchronization targets, synchronizing twice every cycle. *Barrier*-1 and *barrier*-10 are included for reference.

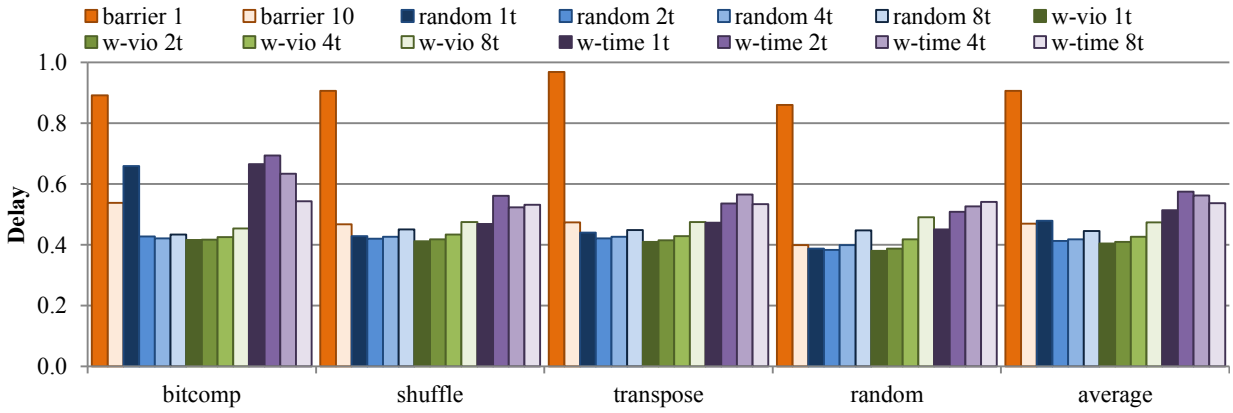


Figure 29: Delay for random and weighted schemes 1, 2, 4, and 8 synchronization targets, synchronizing twice every cycle. *Barrier*-1 and *barrier*-10 are included for reference. Values normalized to *barrier*-0.5.

4.5 CONCLUSION

This chapter introduced weighted-tuple synchronization. Weighted-tuple synchronization is a lightweight, distributed synchronization scheme with two components to improve accuracy over random-pair synchronization. First, tuple synchronization generalizes the synchronization thread granularity beyond one thread used by random-pair, allowing threads to synchronize with multiple targets. The synchronization tuple increases the probability of selecting a good synchronization target. Second, synchronization targets are selected with varying weights with two heuristics evaluated in this work. Because most violations are triggered by a small number of cores, a thread can synchronize more precisely by weighing the probability of selecting a synchronization target to favor cores which trigger the most violations. In addition, a thread can synchronize using the difference in simulated time to synchronize with the slowest threads. The resulting new scheme, *weighted-tuple synchronization*, improves accuracy to be competitive with barrier synchronization while retaining a significant performance advantage.

Weighted-tuple synchronization is experimentally evaluated for both parallel multicore simulation and parallel NoC simulation against both barrier and random-pair synchronization. Tuple synchronization significantly improves accuracy at the cost of some performance, especially at smaller synchronization intervals. For weighted target selection, multicore sim-

ulation benefits the most from weighing by violations because accesses between cores are point-to-point. *Weighted-violation T4* synchronization outperforms barrier synchronization at a synchronization interval of 100 in both accuracy and performance, with a 41% lower average error×delay.

For NoC simulation, flits pass from tile to tile and are affected by the synchronization of multiple threads. Again, tuple synchronization significantly improves accuracy; unlike with multicore simulation, very little performance penalty is observed from adding synchronization targets. For the NoC simulation setting, weighted-time synchronization was found to be the superior target selection technique. Using clock skew as the target selection heuristic, weighted-tuple was shown to be more accurate and faster than the loose synchronization scheme with the best accuracy, *barrier*-1. Compared to the *barrier*-0.5 baseline, *weighted-time T8* synchronization showed an 0.3% average network latency error while being 42% faster.

5.0 RECIPROCAL ABSTRACTION FOR CO-SIMULATION

With growing interest in manycore architectures, researchers often need to study a specialized uncore or off-chip component. State-of-the-art core simulators generally do not model these components in detail or with acceptable performance. Similarly, specialized component simulators do not model the cores in detail. Researchers interested in modeling both must integrate the two simulators; however, integration is often not possible for parallel simulators or when simulators use varying levels of abstraction.

Simulators model hardware elements at varying abstraction levels as described in Section 2.3. Often, simplified core models use multi-cycle time advancement and calculate instruction latencies atomically. In addition, parallel simulators which use loose synchronization also have multi-cycle time advancement from the perspective of other threads. A simulator using multi-cycle time advancement cannot be directly integrated with a cycle-level simulator; the multi-cycle simulators model one instruction at a time, which prevents the cycle-level simulator from modeling contention between multiple requests generated by different instructions.

Because direct integration is not always possible, it is common practice to develop the two architectures independently [42]. This involves feeding a traffic trace from one simulator into the other. For example, a general architecture simulator such as GEM5 [7] or Sniper [9] is used to generate a trace of network or main memory accesses. The trace is then fed into a dedicated simulator such as Hornet [43] or Booksim [34] (for network simulation); or DRAMSim2 [66] (for memory simulation). A key limitation of traces is that they are static; the timing between trace items is not affected by, and does not accurately reflect, the latency of trace item requests.

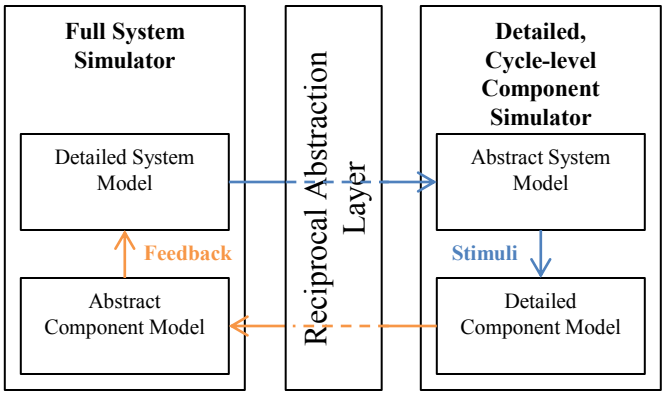


Figure 30: Overview of reciprocal abstraction for co-simulation. Each simulator contains an abstract model of the other with stimuli and feedback being sent through a reciprocal abstraction layer.

This chapter describes *reciprocal abstraction*, which enables accurate co-simulation of computer architecture components modeled with different abstraction levels. An overview of reciprocal abstraction is shown in Fig. 30. Each simulator relies on abstract modeling for a component it does not model in detail; the abstract model has a corresponding detailed model handled by the other simulator. The *reciprocal abstraction layer* provides a bridge between the two simulators to translate the data from each detailed simulator to be used in the corresponding abstract model of the other. This reciprocal abstraction layer also provides the means to accommodate differences in the abstraction levels of the detailed simulators comprising the co-simulation. This work was originally published in [50].

Reciprocal abstraction improves accuracy when co-simulation targets have different abstraction levels. When this is the case, direct integration is not feasible (see Sec. 5.1 for further details). When components can be directly integrated, reciprocal abstraction does not provide an accuracy improvement. However, in Chapter 6, the potential performance benefit from reciprocal abstraction is explored—reciprocal abstraction can thus be applied when co-simulated components operate on the same level of abstraction for improved performance.

The accuracy benefits of reciprocal abstraction are demonstrated in the setting of a manycore architecture with an on-chip network, where a parallel full-system simulator using a simplified core model and abstract network model generates traffic for a cycle-level

NoC simulator. A trace-driven approach is compared with reciprocal abstraction. As a consequence of using a simplified core pipeline model, this full system model uses "atomic" performance estimation where the impact of resource utilization is computed all at once and considers resource contention only indirectly. Moreover, as the full system simulator employs loose synchronization, it is likely that resource requests occur out-of-order from how they would have occurred in a cycle-level simulator. While, in this dissertation, this situation is studied in the context of on-chip networks, it can apply equally well to high-fidelity implementations of other system components such as caches, main memory, or I/O.

5.1 CHALLENGES WITH DIRECT INTEGRATION

In order to accurately model core and network models, integration is required. Serial, cycle-accurate simulators like M5 [8] and Garnet [1] can be directly integrated, but are also very slow when considering hundreds of cores in a target system—simulating one thousand cores for one second would take a year to complete [67]. Simulators which take advantage of core pipeline abstraction or loose synchronization can simulate manycore architectures in a reasonable amount of time. However, such simulators are unable to directly integrate detailed network models.

The first challenge preventing direct integration between the core and network models is the abstraction of the core model pipeline (described in Section 2.3). Instead of exactly simulating the reorder buffer and pipeline state, the cost of each instruction is used to approximate the state of the reorder buffer and track the number of in-flight instructions. However, this means that each instruction needs to determine its full latency atomically, including its network latency. Atomic instruction processing by the core simulator conflicts with cycle-accurate network models, where packets must remain active in the network from injection time to ejection time in order to correctly simulate contention. Furthermore, a detailed network model should model contention between multiple outstanding requests; with atomic instruction processing each core can have at most one active request.

Simulators which use a binary instrumentation front-end for functional simulation also tend to use atomic instruction processing [9, 48, 67]. When the timing model simulation occurs within instrumented code, each instruction's timing must be resolved before the next instruction is simulated. As with abstract core models, an instrumented timing model leads to atomic instruction processing which prevents direct integration with cycle-accurate contention models. The Manifold simulator [76] also uses a binary instrumentation front-end, but buffers decoded instructions for processing by dedicated timing simulation threads. This allows multiple instructions contend with one another in the timing model. However, when functional simulation results are buffered for the timing simulation, an incorrect control flow can arise when the two models disagree on event orderings [15]. Specifically in the case of Manifold, race conditions may be resolved differently when modeling multi-threaded programs—for example, a thread may enter a critical region immediately after a long latency instruction which would have normally led to another thread entering the critical region first.

The second challenge for direct integration is the different synchronization granularities used by the separate parallel simulators. It has been demonstrated that relatively loose synchronization tends not to introduce significant errors for core simulation [9,13], primarily because the communication time between cores is, at minimum, the L1 access latency plus the L2 access latency plus network travel time. For the NoC, however, flits typically traverse links in a single cycle. In these situations, finer-grained synchronization is necessary for parallel NoC simulators. Manycore simulators using dynamic instrumentation and loose synchronization consider synchronization intervals of hundreds or thousands of cycles [9, 48, 67]. A parallel NoC simulator, Hornet [43], studies shorter synchronization intervals of 1-50 cycles due to the reduced lookahead time in NoC simulation. Integration between the two models would require them to use the same synchronization interval; if events can occur anywhere within the larger synchronization interval the smaller synchronization interval loses meaning. Matching synchronization granularities would lead to either a drop in performance for the core model or a drop in accuracy for the network model.

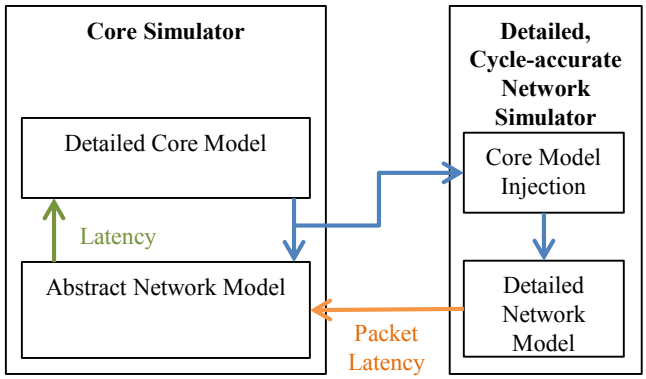


Figure 31: Illustration of co-simulation of core and network simulators.

5.2 RECIPROCAL ABSTRACTION FOR CORE AND NETWORK CO-SIMULATION

The reciprocal abstraction framework for co-simulation of a parallel core simulator with a network simulator is shown in Fig. 31. For the core and network case studied in this work, both the abstract and detailed network models receive the same input (network packets). The core simulator relies on an abstract network model when the memory hierarchy generates network messages for packet latencies. Packets are placed into a trace buffer, which acts as an abstract core model for the network. The network simulator processes packets from the trace, and later updates the abstract network model with statistics regarding interval packet latency for more accurate latency estimates by the abstract model.

To integrate the core and network timing models, simulation is broken up into time step intervals. Simulation for each interval follows these steps, shown in Fig. 32:

1. The core simulator runs while relying on the abstract network model for packet latencies.
2. A network traffic trace is saved in a buffer.
3. The core simulator is paused and control is switched to the detailed network simulator.
4. The network simulator simulates the trace, which represents an abstract core model.
5. The network simulator finishes and the core simulator takes back control.
6. Network statistics from the previous interval update the abstract network model.

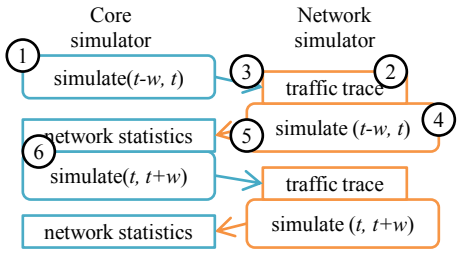


Figure 32: Feedback-driven co-simulation alternating between core and network simulators.

5.2.1 Network Traffic Trace

As the core simulator's abstract network model estimates packet latencies, it generates trace events which act as stimuli for the detailed network model. Core traffic can be heavily skewed in multi-threaded programs; the trace event buffer between the core and network models was designed to handle uneven traffic by using a set of linked lists shown in Fig 33. The trace buffer is divided into many smaller trace chunks; initially, each core thread reserves a single chunk and writes trace items sequentially into the chunk. When a chunk is filled, a free chunk is found and reserved—the end of the filled chunk acts as a tail pointer pointing towards the newly reserved trace chunk. A trace chunk is determined to be free if the last trace item's timestamp is less than the elapsed time of the network model. During network simulation, each tile's injector processes trace items sequentially until it reaches the end of a chunk, at which point it follows the tail pointer to a new trace chunk.

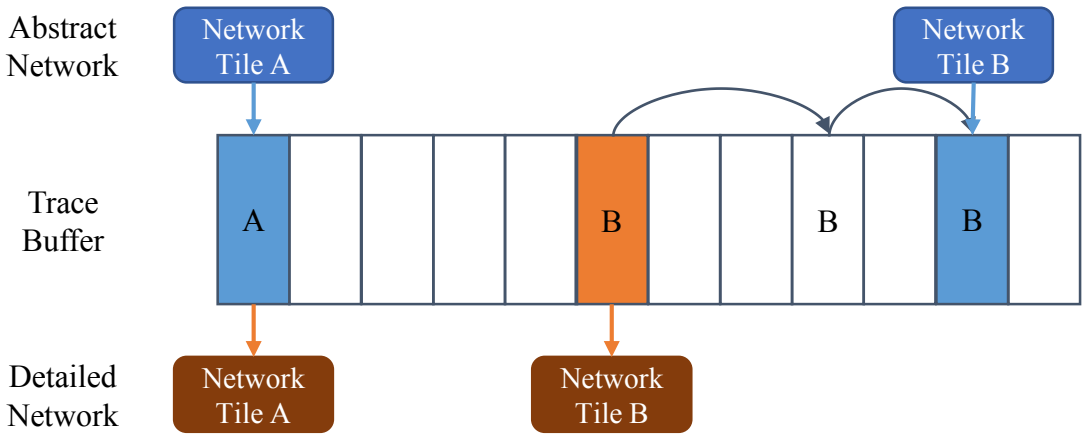


Figure 33: Trace buffer structure used for Reciprocal Abstraction.

5.2.2 Network Simulation

After executing for an interval $(t, t + w]$, the core simulator checks whether it should switch to the network simulator. This occurs when all cores have advanced their simulation time since the previous network interval, in which case the network model is set to simulate all events up to the minimum core simulation time. However, cores can become blocked due to system calls and thus fail to advance time even after executing the barrier. To prevent the core model from executing for too long, if core simulation advances beyond the network simulation by some threshold, a switch is forced. The network simulator is then directed to simulate until it catches up to the core simulation.

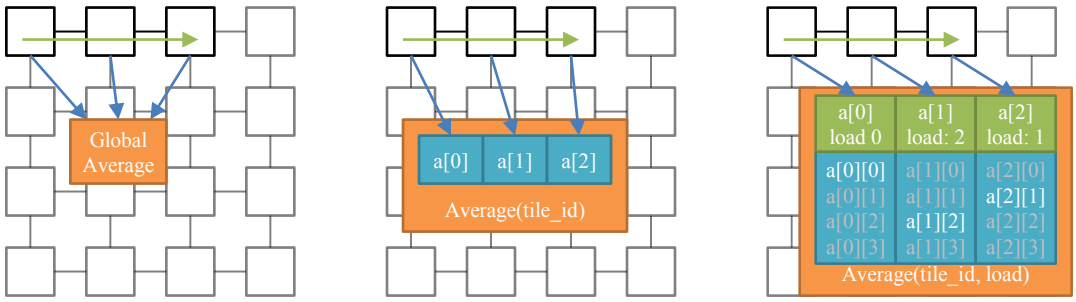
When the network model is switched to, all core threads are paused and the network threads simulate the trace items in its specified window of cycles. Network simulation proceeds it would in a standalone situation, although additional statistics are stored for the window being simulated, which are used to update the abstract network model once detailed network simulation completes.

5.2.3 Feedback Update Strategies

Within the general feedback-driven framework described above, the detailed network model can update the abstract network model using different strategies. As a baseline, each network router uses the most recent average latency for flits traversing the router during detailed simulation.

There are two primary considerations for updating an abstract model's latency. The first is what form of temporal prediction to use. Given average latencies, L_1 and L_2 for intervals I_1 and I_2 respectively, the average latency L_3 for I_3 is predicted as one of the following:

- *previous*: Use the most recent information, set the latency to that measured in the previous interval: $L_3 = L_2$
- *moving average*: Use a value in between the two most recent intervals, set the latency to a moving average: $L_3 = avg(L_1, L_2)$
- *linear*: Assume latency will continue its upward or downward trend, make a linear prediction: $L_3 = L_2 + \Delta(L_1, L_2)$, where $\Delta(L_1, L_2) = L_2 - L_1$



(a) Global average hop latency

(b) Per-router average hop latency

(c) Per-router load-based average hop latency

Figure 34: Prediction function specificity. In 34a, a single average hop latency used for the entire network. In 34b, each network tile uses its own average hop latency. In 34c, each tile uses a table indexed by packet load.

The second consideration is how specifically to treat packets—how many separate prediction functions should be used? These schemes are illustrated Fig. 34. At one extreme the abstract model can maintain a single average hop latency for the whole network (Fig. 34a). Another option is to break down packet travel by hop and keep separate prediction functions for each router (Fig. 34b). Load-latency curves [57] can provide more specificity, further breaking down flits traversing a router based on the router's current load (Fig. 34c).

For load-latency curves, load is defined as the number of packets which have passed through the router recently, and is separated into bins for a table lookup. Packet load is tracked in windows of 64 cycles as in [57]. For this work, the core timing model does not model packets in order, so load from one model does not translate well to the other. Because of this, load-latency table lookup uses load measured by the abstract network model—this load is passed to the detailed network model and used to calculate the average hop latency for the appropriate load bin. The abstract model is a window-based contention model (see Fig. 2)—the number of packets which arrived in the load window is calculated by dividing the busy time by the packet occupancy time (3 cycles). For evaluation, the per-router average (Fig. 34b) and a load-based table lookup (Fig. 34c) were used.

5.2.4 Generalizing Reciprocal Abstraction

Reciprocal abstraction is generalizable to other simulated components besides an on-chip network. There are two major component-specific considerations for applying reciprocal abstraction to a component model:

- The abstract component model used by the full-system simulation and the necessary feedback to train it.

- The stimulus from the full-system simulation to the detailed component simulation.

A variety of abstract component models have been researched to improve simulation speeds, which are described in Section 2.3. Categorically, they are either empirical (no knowledge of architecture) or mechanistic (assumes knowledge about the underlying architecture). For this work evaluating core and NoC co-simulation, the network model uses an empirical contention model, where the hop latency is trained over time. Other contention-based structures can also use an empirical contention models; these include memory controllers and network-on-chip routers. Mechanistic abstract models are useful when a hardware structure's latency is dependent on specifics in the workload such as memory addresses. StatStack [22] presents a mechanistic model for an LRU cache which depends on address reuse distance. Similarly, a mechanistic model can be used for branch prediction, where hits and misses depend on the branch instruction address, as presented in [23].

The stimulus from the full-system simulator to the component simulation is most often a trace of accesses or requests. For NoC simulation, packet trace information includes the source, destination, size and injection time. For memory accesses, an address and timestamp may be sufficient. Some abstract models can be trained on only a subset of the total requests. For example, in the cache model StatStack [22], cache accesses are sampled hierarchically. Sampling is effective when a component can be partition and the behavior for one partition can be generalized to others. Sampling is not as effective for contention models, where all accesses generate contention.

5.3 EXPERIMENTAL EVALUATION

To evaluate reciprocal abstraction, this work uses a core simulator coupled with a network-on-chip simulator. First, the inaccuracy of a trace-based approach is established. To measure error, the simulators are run in isolation and the network packet latencies assumed by the core simulator's abstract model and those simulated in detail by the network are compared. This work then shows that the change in network latencies would have, in turn, impacted the core model's performance and traffic injection rate.

Once the inaccuracy of isolated simulation is established, the benefit of reciprocal abstraction is demonstrated. When both simulators are run loosely integrated under the reciprocal abstraction framework, the difference in network latency is significantly reduced. The resulting changes to the core simulator's cycles-per-instruction (CPI) and injected packets is demonstrated.

Two L2 cache organizations are evaluated: private and shared. In the private configuration, each core has a private L2 cache; the cache slices are kept coherent using the MESI protocol. For private caches, network traffic consists of coherence traffic and traffic to the interleaved memory controllers. Much of the coherence traffic is not on the critical path; thus it adds to network traffic but does not directly impact performance.

A distributed shared or NUCA cache [36] is also evaluated. Under a shared configuration, all L2 caches logically form a single cache. Physically, the address space is interleaved at the block level amongst the cache slices. In this configuration, there is no L2 coherence traffic, but all L2 accesses may need to access the network. Thus, there is more network traffic and network traffic has a greater impact on core performance.

5.3.1 Setup

For evaluation, the Sniper parallel multicore simulator (see Section 2.4.2) is used for the core simulator, and the Hornet parallel network simulator (see Section 2.4.3) is used for the network simulator. Upon initialization, Sniper launches a pool of threads which act as Hornet threads. Sniper threads synchronize every 100 cycles, while the Hornet threads synchronize

twice a cycle (at each "positive" and "negative" clock edge). The loosely-integrated simulator switches between core and network modes at barrier boundaries so only one simulator has active threads at a given time.

Hornet was modified from the release version to work with reciprocal abstraction as detailed in Sec. 5.2. In addition, several modifications were made to reduce its memory footprint. By default, Hornet relies on flow tables for routing and channel allocation. However, for a network with N tiles, there are N^2 flows (one for each source, destination pair), which results in large routing tables. For this evaluation, the tables were replaced with routing functions—basic routing functions for dimension order and O1TURN [68] routing were implemented. In addition, several structures which were re-allocated every cycle were modified to be allocated only once.

The target machine used for evaluation is a 256-core machine, with cores arranged on a 16x16 mesh. Host and target machine specifics are listed in Table 5. Benchmark details can be found in the Appendix, Table 7. All benchmarks are run for a maximum of 10 million cycles per core, which is equivalent to 160 million cycles for a 16-core machine.

5.3.2 Inaccuracy of Using a Component Simulator in Isolation

This section shows that the detailed simulation of either the core or network when using an abstract model of the other in isolation leads to inaccuracy. First, core simulation is performed using an abstract network model in isolation; a trace of generated traffic is then fed into a detailed network model. Experiments show that the trace of generated traffic can lead to saturation. After getting statistics from the detailed network simulator; the calculated packet latencies are fed back to the core simulator to show the impact of network latencies on core CPI. This CPI change would, in turn, impact network traffic generation. Thus, the trace-driven approach where both simulators are run in isolation eliminates an important feedback loop required for accurate simulation.

The average hop latency generated by the multicore simulator's abstract network model (using the contention model described in Section 2.3) was compared against the latency generated by the detailed network model. The trace of network traffic generated when

Table 5: Reciprocal Abstraction Host and Target Machine Parameters

Host Machine	Dual Socket Xeon X5680 @ 3.3GHz (2x 6 cores w/ HyperThreading). 96GB RAM
Core Simulator	
Core	2GHz, in-order commit with out-of-order memory accesses
Memory Hierarchy	32KB, 4-way private L1 Instruction Cache 32KB, 4-way private L1 Data Cache 512KB, 16-way private or shared L2 Cache with GHB prefetcher
Main Memory	Controller's interleaved every 8 tiles 50ns access latency
Network Model	
Layout	16×16 mesh
Pipeline Stages	3
Virtual Queues	4 per link
Virtual Queue Size	8 flits
Routing	X-Y

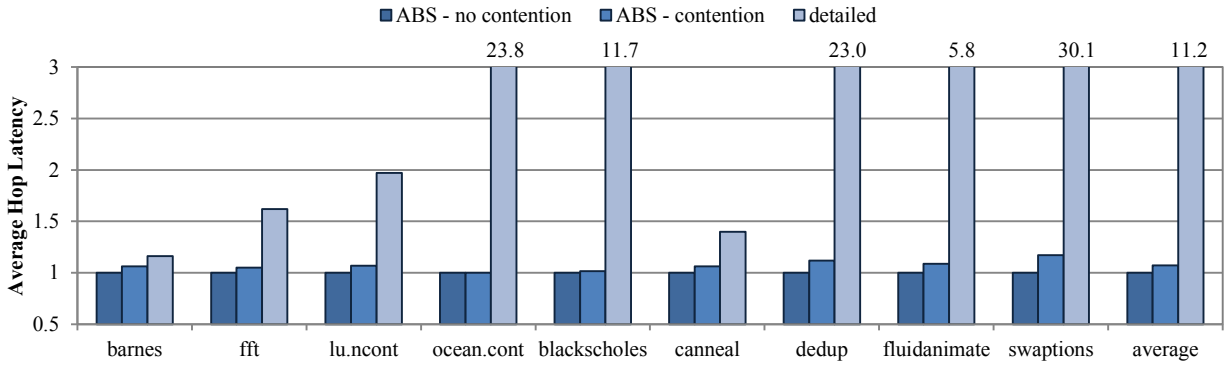
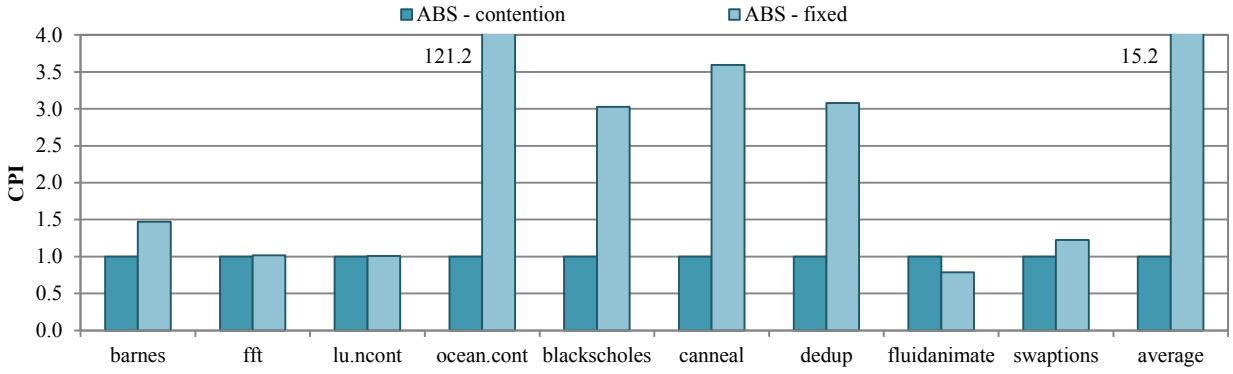


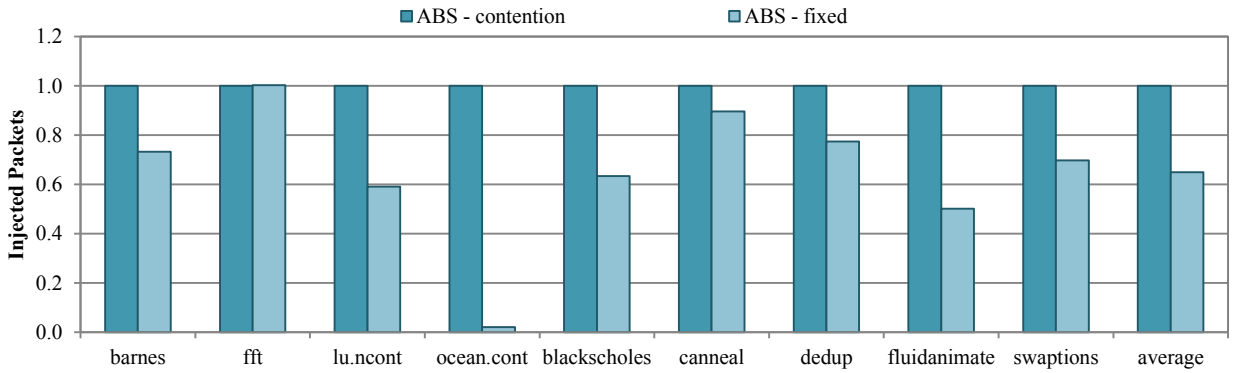
Figure 35: Average Hop Latency for: abstract network model assuming no contention (ABS - no contention), abstract network model using contention model from Fig. 2 (ABS - contention), and detailed network model. Traffic for detailed model is generated by using the traffic trace from **ABS**-*contention*. Simulation uses private L2 caches.

using abstract network model's trace is used as input for the detailed network simulation. First, a private L2 configuration is studied; Fig. 35 shows measured hop latencies for three models: an abstract model assuming no contention for a hop latency of three cycles (**ABS**-*no contention*), an abstract model assuming simple contention (**ABS**-*contention*), and a detailed network model (*detailed*) with traffic generated by the **ABS**-*contention* model. Hop latencies are normalized to hop latencies generated by the **ABS**-*no contention* model. Hop latencies estimated by the abstract model differ substantially from those generated by the detailed model. In some cases, the detailed network simulation becomes saturated; even in benchmarks which do not reach network saturation, average network latencies differ by 45%.

The hop latencies generated by the detailed model from Fig. 35 are then fed back into the abstract network model and the core simulator is rerun using the new network latencies, labeled as **ABS**-*fixed*. Fig. 36 shows the simulated CPI and network injection rate compared to the initial simulation run with the core simulator in isolation. The experiment shows an overcompensation of network latencies, as the generated network traffic has now decreased. In a real system, higher network latencies would have translated into lower core throughput and naturally throttled the network traffic. Network traffic, latency, and core throughput have a cyclical dependency. This interaction is missed when simulating the models in isolation.



(a) CPI



(b) Network Traffic

Figure 36: Average CPI and network traffic generated by the core simulator running with an isolated abstract network model. The abstract models are: contention queue model from Fig. 2 (ABS - contention) and a fixed-latency model using latencies generated by the detailed network model from 35 (ABS - fixed). Simulation uses private L2 caches.

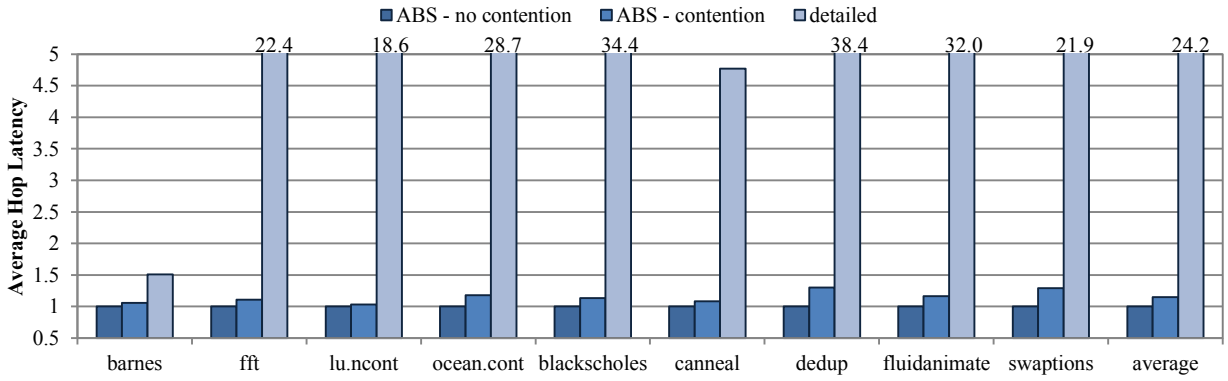


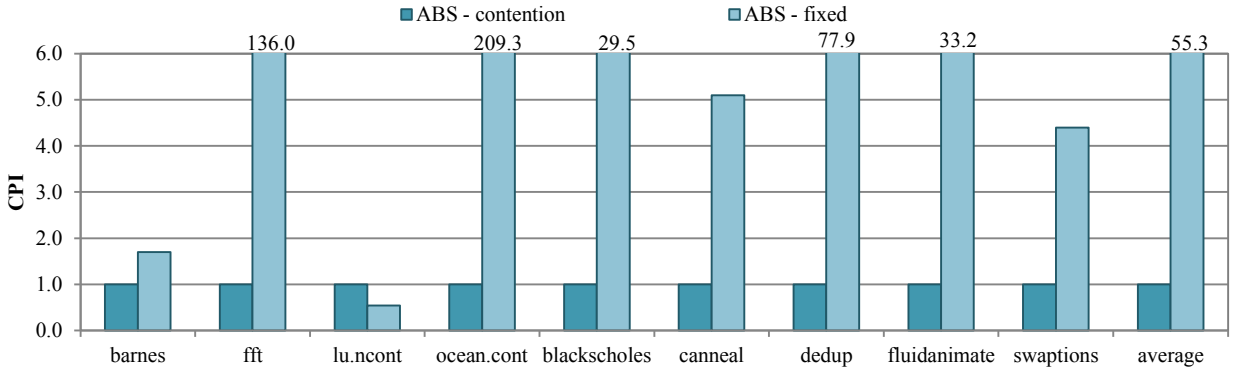
Figure 37: Average Hop Latency for: abstract network model assuming no contention (ABS - no contention), abstract network model using contention model from Fig. 2 (ABS - contention), and detailed network model. Traffic for detailed model is generated by using the traffic trace from **ABS**-*contention*. Simulation uses shared L2 caches.

This experiment is repeated for a distributed shared L2 configuration. The hop latency estimation from the abstract model is compared against the detail model's simulated latency in Figure 37. As seen with private caches, many benchmarks reach saturation when fed a trace generated by the abstract model. Hop latency deviation is greater for a shared configuration compared to private, as more network traffic is generated from L2 accesses traversing the network.

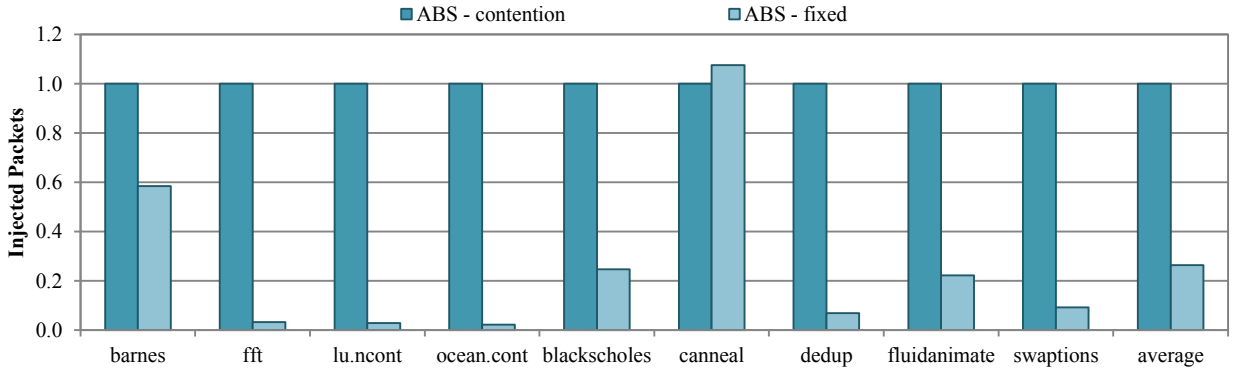
The network latencies from Figure 37 are fed back into Sniper's abstract network model to show the impact isolated simulation has on the core simulation. Average core CPI is shown in Figure 38a and the number of injected packets is shown in Figure 38b. Compared to the private cache organization, core performance is even further from the original simulation using only the abstract model; this is due to both the higher network latency and the greater dependence between core performance and network latency. Modeling the interaction between the core and component models is even more important when the component model heavily impacts core performance, as is the case for a distributed shared cache configuration.

5.3.3 Reciprocal Abstraction Error Impact

To evaluate reciprocal abstraction, network latency error is compared between the isolated simulation approach used in the previous section with reciprocal abstraction. For both



(a) CPI



(b) Network Traffic

Figure 38: Average CPI and network traffic generated by the core simulator running with an isolated abstract network model. The abstract models are: contention queue model from Fig. 2 (ABS - contention) and a fixed-latency model using latencies generated by the detailed network model from 37 (ABS - fixed). Simulation uses shared L2 caches.

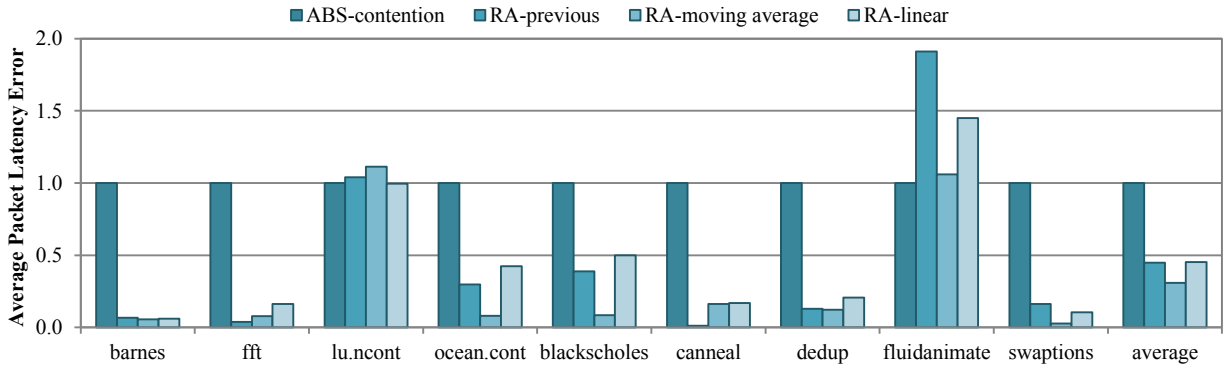


Figure 39: Average packet latency error for **ABS**-*contention*, **RA**-*previous*, **RA**-*moving average*, and **RA**-*linear* schemes. Results are normalized to error for **ABS**-*contention*. Simulation uses private L2 caches.

configurations, a trace of network accesses is passed to a detailed NoC simulator. To measure packet latency error, the estimated latency from the abstract model is compared with the cycle-accurate latency from the detailed model for each packet; the packet's latency error is the difference between the two latencies. The following temporal prediction schemes, described in Section 5.2.3, are compared:

- **ABS**-*contention*: baseline where the core simulator relies on the abstract contention model for all network latencies.
- **RA**-*previous*: Reciprocal-Abstraction approach which uses the *previous* temporal prediction scheme.
- **RA**-*moving average*: Reciprocal-Abstraction approach which uses the *moving average* temporal prediction scheme.
- **RA**-*linear*: Reciprocal-Abstraction approach which uses the *linear* temporal prediction scheme.

5.3.3.1 Reciprocal Abstraction for Private L2 Caches

Fig. 39 shows the average packet latency error comparing temporal prediction schemes, using per-router average latency. The best temporal prediction scheme varies with workload, indicating the need for a more adaptive scheme. For fluidanimate, latency estimation is thrown off by latency spiking and receding more quickly than the interval size.

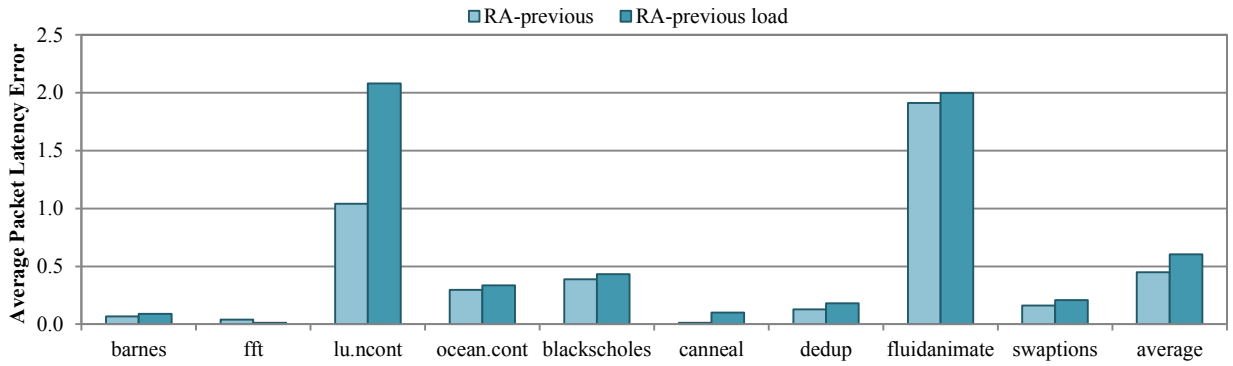
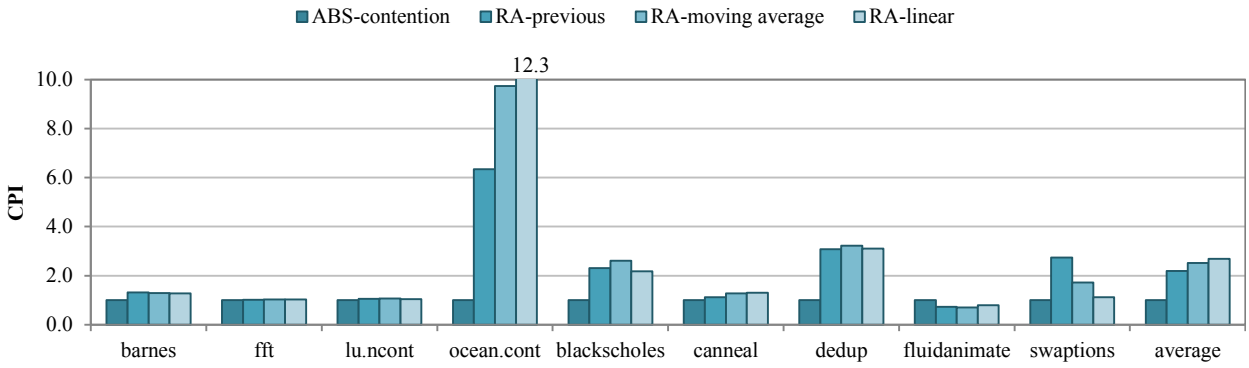
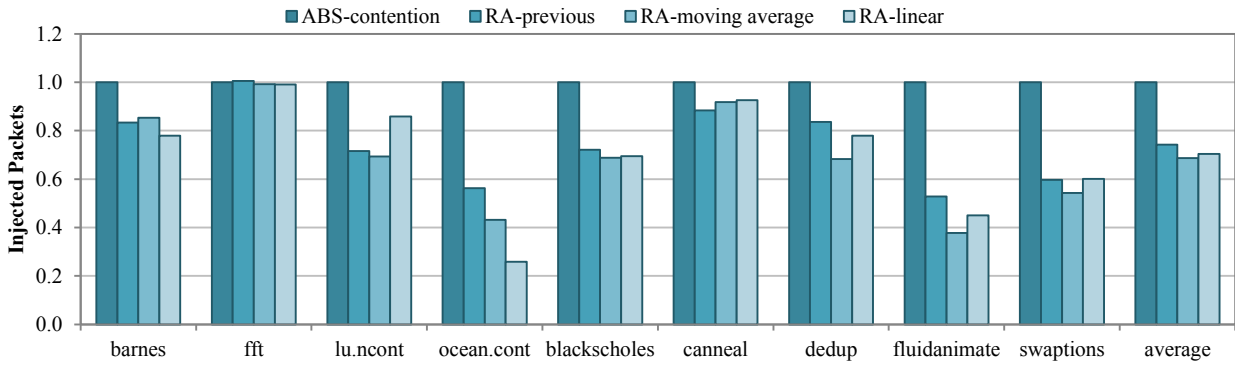


Figure 40: Average packet latency error with and without load-based packet classification, using the *previous* temporal update strategy. **RA**-*previous* uses a single latency value, while **RA**-*previous load* uses load-latency curves. Error is normalized to core simulator with isolated abstract network model assuming simple contention model (**ABS**-*contention*). Simulation uses private L2 caches.



(a) CPI



(b) Network Traffic

Figure 41: CPI and packet injection rate with *previous interval*, *moving average*, and *linear* latency update strategies. Values for both CPI and injection rate are normalized to core simulator with isolated abstract network model assuming simple contention model (**ABS**-*contention*). Simulation uses private L2 caches.

The use of load-latency curves for reciprocal abstraction is evaluated; Fig. 40 illustrates the results. Categorizing packets based on their load, where load is generated by the abstract network model, does not improve latency estimation. This is because the measurement of load for the abstract model is inaccurate due to out-of-order packet generation and inexact network timestamps.

Next, the cascading effects correcting packet latencies have on the rest of the simulator are explored. The evaluation metrics for accuracy are the core simulator's CPI and the network simulator's injection rate. Fig. 41a shows the impact the feedback-driven network model has on core CPI. After adding network feedback, core CPI changed by an average of 150% compared to the no-feedback abstract network model. This highlights the importance of an accurate network model when studying manycore architectures. Correcting the packet latencies and CPI of the core simulator in turn impacts the network's injection rate, which is shown in Fig. 41b.

5.3.3.2 Reciprocal Abstraction for Shared L2 Caches

The effect of reciprocal abstraction is also evaluated for a distributed shared cache configuration. In Fig. 42, the average latency deviation between the abstract and detailed network models is compared for reciprocal abstraction using various temporal prediction schemes; the isolated **ABS**-*contention* model serves as a baseline. The reduction in packet latency error is more pronounced for shared caches. Reciprocal abstraction reduces the packet latency error by 81%, 79%, and 80% for the *previous*, *moving average*, and *linear* prediction schemes, respectively. The temporal prediction scheme does not significantly affect the hop latency error, implying that the traffic level does not change heavily in the distributed shared setting.

The impact reciprocal abstraction has on core performance and network traffic is shown in Figure 43a. Adding feedback between the core and network models dramatically affects the core model's CPI and network traffic injection rate. Compared to the private cache configuration, where CPI changed by 150% on average, with shared caches reciprocal abstraction results in a CPI nearly five times as high as an isolated simulation with no feedback. The reduced core throughput results in fewer injected packets, as shown in Figure 43b.

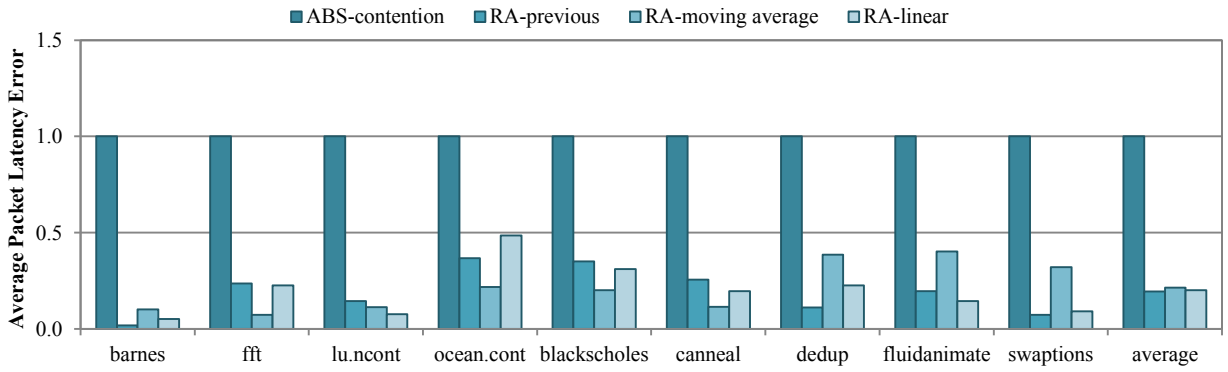
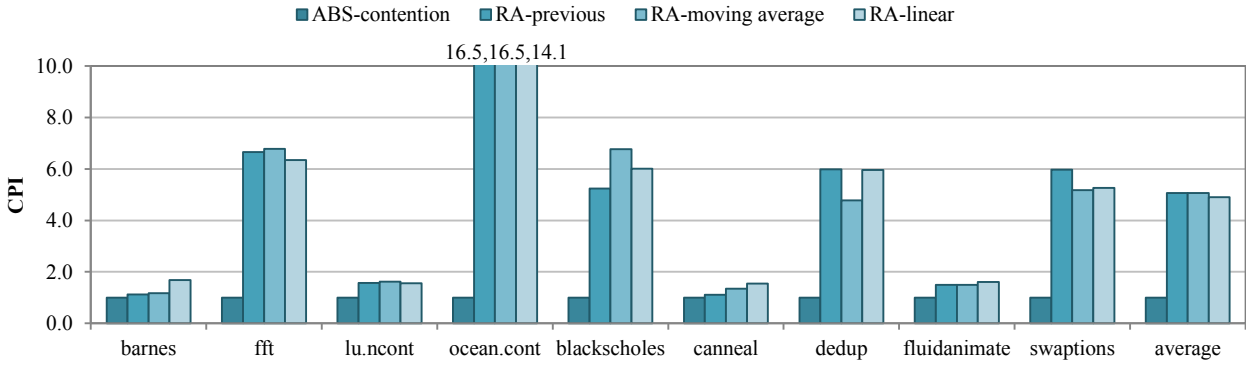
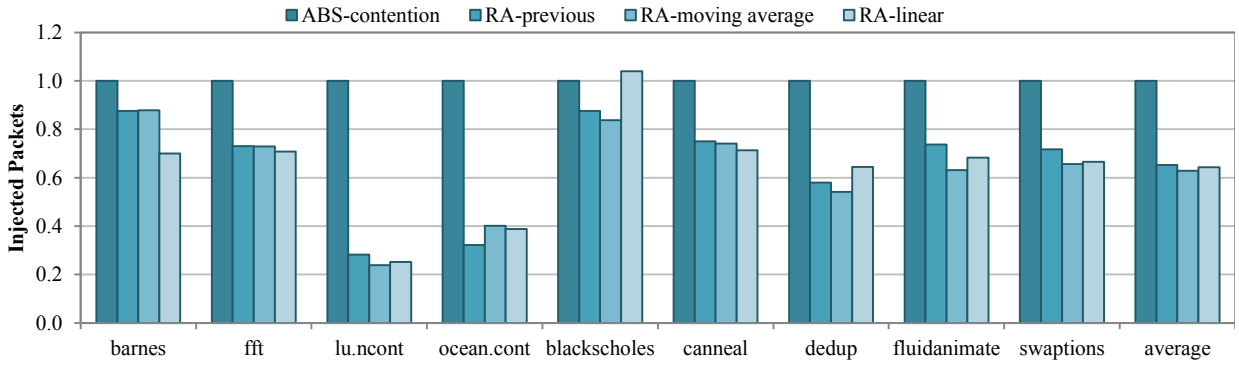


Figure 42: Average packet latency error for **ABS**-*contention*, **RA**-*previous*, **RA**-*moving average*, and **RA**-*linear* schemes. Results are normalized to error for **ABS**-*contention*. Simulation uses shared L2 caches.



(a) CPI



(b) Network Traffic

Figure 43: CPI and packet injection rate with *previous interval*, *moving average*, and *linear* latency update strategies. Values for both CPI and injection rate are normalized to core simulator with isolated abstract network model assuming simple contention model (**ABS**-*contention*). Simulation uses shared L2 caches.

5.4 CONCLUSION

In this chapter, co-simulation between multiple component simulators was studied. Often, two simulators can not be directly integrated due to differences in their level of abstraction and synchronization granularity. Using a core simulator and network-on-chip-simulator, this chapter demonstrated significant error in packet latency when the core model relies on an abstract NoC model in isolation. Deviation in CPI and packet injection rate further shows that using a static trace for modeling either component in isolation will lead to inaccurate results.

Reciprocal abstraction was proposed to allow for accurate integration; under reciprocal abstraction, a simulator's detailed model maintains an abstract model of the other simulator. The simulators periodically exchange detailed statistics to update the abstract models. After applying reciprocal abstraction, the integrated simulator reduced packet latency error compared to using a simplified, abstract NoC model in isolation. Packet latency error is reduced by over 69% for private L2 caches and over 79% for shared L2 caches. Reciprocal abstraction is shown to model the feedback loop for real systems, where network latency affects core throughput, which, in turn, affects the rate of network traffic generation.

6.0 PERFORMANCE BENEFITS OF RECIPROCAL ABSTRACTION

The primary goal of reciprocal abstraction is to improve simulation accuracy by loosely integrating two component simulators which could not otherwise be integrated. However, another benefit of reciprocal abstraction is that the two simulators can run in parallel, with concurrent threads modeling heterogeneous components. Conventional parallel simulation divides work amongst homogeneous simulator threads; this chapter investigates the alternative parallelization model and its performance implications. The relaxed communication requirements allow one component simulator to be offloaded to a coprocessor. Work from this chapter was published in [50].

6.1 COPROCESSOR ACCELERATION

With reciprocal abstraction, the full-system and specialized component simulators communicate infrequently. The detailed component model only needs to read the trace buffer before it simulates an interval, and the core model only needs to update its abstract component model after its corresponding detailed model simulates an interval. By offloading one simulator to a coprocessor, the detailed simulation of each interval can be pipelined. By overlapping simulation of both models, the overall simulation time is reduced. This approach is demonstrated in the setting of core and network-on-chip simulation; a GPU is used to simulate a detailed network model concurrently with the host CPU simulating the core model.

6.1.1 General Purpose Graphics Processing with CUDA

Using GPUs for general purpose processing was made popular with NVIDIA's CUDA technology [17]. GPUs, originally designed for games and video processing, are massively parallel Single-Instruction, Multiple-Thread (SIMT) processing units. Compared to conventional CPUs, GPUs devote more transistors on computation and memory bandwidth and fewer to control and cache structures. This allows GPUs to have much greater computing potential than CPUs, provided a workload exhibits enough parallelism. This section describes aspects of the CUDA programming model relevant to this work; further information on CUDA can be found in [17].

Architecturally, GPU cores are organized in a hierarchy. At the top level, the GPU is composed of a number of Streaming Multiprocessors (SMs). Each SM contains several SIMT units; each made up of CUDA cores which execute instructions in lockstep. The CUDA programming model has a similar hierarchy. A CUDA program launches GPU code as a kernel. Each kernel launch is broken into a number of thread blocks; thread blocks are scheduled onto SMs. The thread block contains a number of warps. Each warp is scheduled to run on one SIMT unit and contains 32 threads; threads in the same warp must execute instructions in lockstep. When a warp encounters a control instruction, threads in the warp may not follow the same branch path. This causes a warp divergence, where threads on one branch path execute while the others sit idle, lowering core utilization.

Threads on a SIMT unit can access memory simultaneously, as follows. Global memory accesses must be *coalesced* in order to occur in parallel—that is, accessed memory must reside in the same 32-word chunk. Uncoalesced memory accesses are serialized.

6.1.2 GPU-based Network Simulation

The GPU-based network simulator evaluated models the same network architecture as the Hornet parallel NoC simulator [43]. The CUDA-based simulator was developed and validated separately before integration for co-simulation. Packets are injected from a trace into all tiles. The tiles then, in parallel, simulate the router pipeline stages: routing, VC allocation,

crossbar arbitration, and switch traversal as described in Section 2.4.3. Each CUDA thread handles the task of simulating one network tile.

As is the case with Hornet, simulation of each clock cycle is broken into two stages, representing the positive and negative edges of the clock cycle. During each stage, data is only modified by a single thread. Therefore, a barrier—inserted after each stage—is sufficient to prevent state violations from synchronization.

A barrier for threads in the same block is supported natively and has very little overhead. Inter-block synchronization is only natively supported by ending kernel execution. With typical simulations lasting millions of cycles, the overhead from starting and stopping a kernel is too high. However, inter-block barriers can be implemented using built-in atomic instructions. Xiao and Feng describe an inter-block barrier where one thread from each block atomically increments a counter and threads spin on the counter until it reaches the target value [81]. The algorithm in [81] is not safely reusable, as deadlock can occur when barriers are executed close to one another. For this evaluation, the barrier was modified slightly to use a sense variable: threads spin on a sense variable and the last thread resets both the counter and the sense variable; this avoids barrier reuse errors because the barrier is reset properly.

Validation: Implementation details and small artifacts prevent the GPU-based simulator from exactly matching Hornet's results. For example, random number generation, which is used during arbitration, has different implementations in the C++ and CUDA libraries. In addition, iteration over certain standard library structures used in Hornet (such as a map) is difficult to reproduce exactly. The GPU network simulator was validated against Hornet prior to integration with Sniper and an average 2.28% deviation in packet latency was measured for synthetic workloads.

6.1.3 Maximizing GPU Efficiency

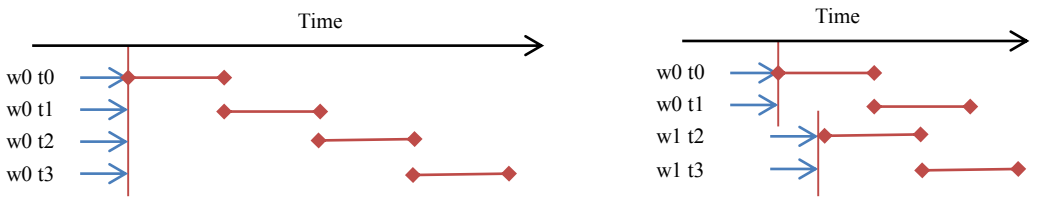
A straightforward CUDA implementation of a network simulator has an irregular memory access pattern which serializes accesses to global memory. Moreover, the virtual queue structures holding flit state do not fit within shared memory. The GPU-based network simulation

creates a performance bottleneck unless optimizations are made to improve performance. In this evaluation, two techniques are used in order improve simulator efficiency on a GPU architecture. First, nested loops are split to reduce divergences containing memory accesses. Second, warps are intentionally underpopulated to increase scheduler latency tolerance and memory concurrency.

Loop Splitting: As SIMT processors, GPU IPC is maximized when all threads in the warp execute the same instruction. When threads have *divergent* instruction paths, execution is serialized and reduces performance. One cause of divergences is the use of nested loops where the inner loops is conditional, as it is unlikely all threads will execute the same set of inner loops. Where possible, the network simulator code has nested loops split up to reduce divergent code. In a loop where work is done for some elements that meet a condition, these elements are first put into a list. This list is iterated over separately. A common process that occurs during simulation is checking all ingress queues for head flits that need virtual channel allocations. On a GPU, branch divergence overhead is reduced when the simulator first gathers a list of ready virtual channels, then iterates over the secondary list to process them.

Underpopulated Warps: Unlike multithreaded CPU applications, GPU applications typically launch many more threads than there are GPU cores. Threads are allocated register space that is not swapped out when the thread is not running, removing context switch overhead. Fast thread switching allows threads to be swapped out when they encounter long-latency stalls. In [53], authors propose to modify the thread scheduler to address this issue. Their scheduler breaks threads into two pools with staggered execution to improve overlapping of execution with memory accesses.

When mapping each network tile to a GPU core, there are not enough active threads to swap stalled threads out. In addition, early investigations found many memory accesses in the GPU-based simulator were difficult to coalesce—limiting parallelism within a warp. This leads to serialized thread execution as shown in Fig. 44a (a hypothetical example with four threads per warp). Memory accesses within the warp cannot be executed together and become serialized. Without other warps to swap in latency for memory accesses cannot be hidden.



(a) One warp with four threads

(b) Two warps, each with two threads

Figure 44: GPU behavior when memory accesses are not coalesced and occupancy is low.

To deal with the low number of threads, warps were intentionally underpopulated by a *thinning* factor of K. Instead of launching a N warps (with 32 active threads each), $N \times K$ warps are launched and, via a thread index check, only $32/K$ threads in each warp have work assigned to them. Thinning out warps leads to more active warps and allows them to hide memory latencies; once each underpopulated warp stalls at a memory access, the next warp can be swapped in. Fig. 44b shows a hypothetical example where the four threads in one warp have been split into two active threads per warp. Increasing the number of warps in this manner is an artificial way to add scheduling latency tolerance, which was shown to improve performance in [53].

6.1.4 GPU-Accelerated Reciprocal Abstraction

The GPU-based network simulator just described was integrated into the Sniper multicore simulator described in Section 2.4.2. As in CPU-only reciprocal abstraction, core simulator threads rely on the abstract network model and place network traffic into a trace buffer. The trace buffer is mapped to zero-copy memory, which is copied to the GPU in the background.

After each reciprocal abstraction interval, the core simulator invokes the network simulator on the GPU. The core simulator does not wait for the kernel to complete, and instead continues simulation. After the next interval, the core simulator makes sure the previous network simulation kernel has completed, waiting if necessary, before starting the next interval of network simulation. This allows both core and network models to run in parallel, with execution is staggered slightly from that shown in Fig. 32. The core simulator now delays its

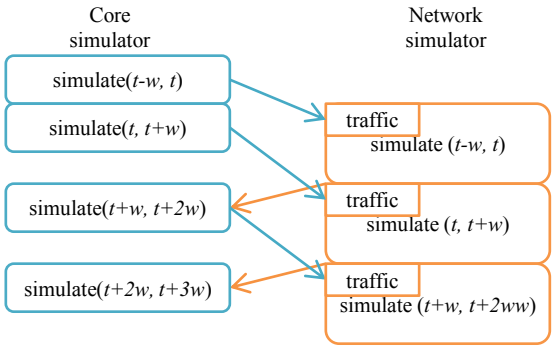


Figure 45: Reciprocal abstraction co-simulation with core and network simulators executing concurrently.

Table 6: GPU Acceleration Host Machine

Name	Processor	CPU cores / GPU SMs	CUDA cores
CPU	Intel Core2 Duo E8400	2	NA
GPU	NVIDIA Tesla C2075	14	448

abstract model update by one interval as shown in Fig. 45. While model update is delayed one interval, with short interval lengths this did not have a significant impact on accuracy.

6.1.5 Experimental Evaluation

Simulations use the same target machine and benchmarks described in Section 5.3.1. Table 6 lists the host machine setup for evaluation of coprocessor acceleration using a GPU. The serialized timing simulators running only on the host's CPU is compared with the parallel timing model running on the same system utilizing an attached GPU. For these performance evaluations, runs were limited to 1 million cycles.

Fig. 46 shows simulation time when utilizing a GPU coprocessor. The simulation time of coprocessor-accelerated simulation, **CA**, is broken down into time spent in the core simulator and time spent waiting for the network model. Warps are launched with 8 active threads rather than 32 to increase GPU occupancy as described in 6.1.3—this reduced simulation time by 36% over launching full warps for a 256-core target machine. On average, utilizing the GPU coprocessor reduces simulation time by 16%. For some benchmarks, the network

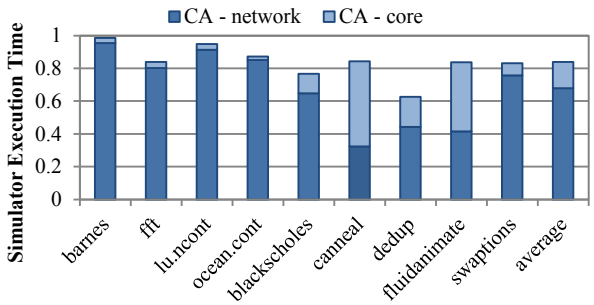


Figure 46: Simulation time with coprocessor acceleration simulating a 256-core system. Time is normalized to simulation time with serialized core and network simulation, and broken down between the core and network models—overlapped execution counts towards core simulator time.

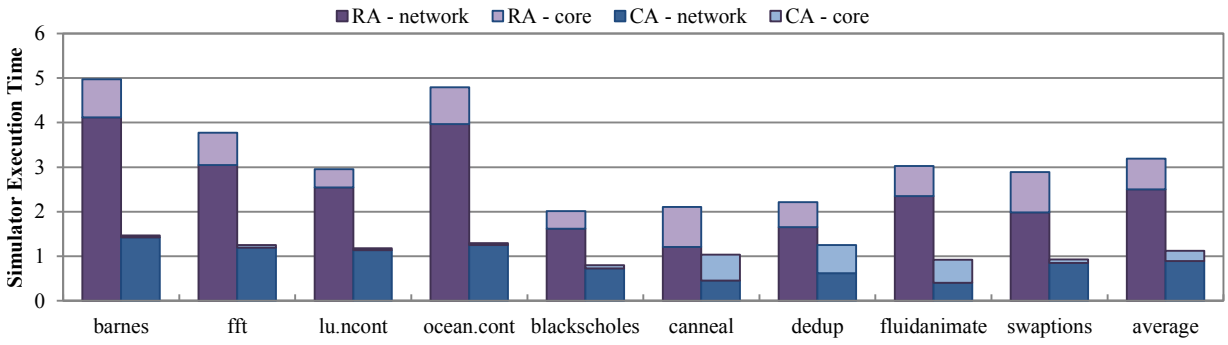


Figure 47: Simulation time with coprocessor acceleration simulating a 512-core system. Time is normalized to simulation time with serialized core and network simulation for a 256-core system to show time scaling from 256 to 512 cores. Simulation time is broken down between the core and network simulation—overlapped execution counts towards core simulator time.

model dominates simulation time. In this case, simulating both models in parallel grants no benefit and performance is determined by the network model simulator's performance.

The scalability of the GPU-based network simulation is demonstrated by simulating a 512-core system with cores arranged on a 32:16 mesh in Fig. 47; to show scaling from 256 to 512 cores, simulation time is normalized to the CPU-based integrated simulator, **RA**, modeling 256 tiles. The left bars represent the simulation time for **RA** when modeling 512 tiles. Simulation time increases by over 200% on average from 256-tile simulation due to increased memory pressure and contention for host cores. The right bars show the simulation time with GPU acceleration when modeling 512 tiles. Because network modeling time increases sublinearly on the GPU and superlinearly on the CPU, the GPU-accelerated simulator models a 512-core system 65% faster than the CPU-only simulator. Based on these measurements, GPU-based acceleration offers the best results when:

- There are enough simulated threads to maximize GPU thread occupancy.
- The core simulation makes up a large enough fraction of overall simulation time, such that there is significant overlap between the core and network simulation.

6.2 CONCLUSION

In this chapter, the performance implications of reciprocal abstraction were evaluated. Due to the asynchronous nature of the two models when using reciprocal abstraction, the approach can be accelerated by running both models concurrently. In addition, using a coprocessor, one model can be offloaded—this work described a CUDA-based detailed network simulator GPU implementation and validated it against a widely-used detailed network simulator [43]. With the integrated simulator, the GPU-based simulator reduces simulation time by 16% and 65% on average for 256 and 512-core target machines, respectively.

7.0 CONCLUSIONS

For my thesis, I studied parallel simulation of computer architectures, where synchronization is a key factor in determining performance and accuracy. I first analyzed popular parallel simulators and their synchronization techniques. My analysis of violations both enables the rest of my work and contributes to future research into loose synchronization policies.

After examining barrier and random-pair synchronization, I developed a novel synchronization technique, weighted-tuple synchronization. Weighted-tuple synchronization synchronizes a subset of the simulation threads, resulting in lower overhead compared to barrier synchronization and higher accuracy compared to random-pair synchronization. Moreover, weighted target selection ensures threads synchronize with the candidates most likely to trigger violations. Evaluations using multiple simulation environments show weighted-tuple to be a superior form of loose synchronization compared to barrier synchronization. While this dissertation focuses on computer architecture simulation in its evaluation of weighted-tuple synchronization, the policy has potential application in more general PDES simulations which suffer from limited parallel scalability.

Modern hardware has grown too complex for a single simulator to model all components in detail; thus, research which requires detailed component simulation must combine two or more simulators. The second part of my research investigates parallel co-simulation between multiple simulators responsible for modeling separate hardware components. I introduce the reciprocal abstraction framework, which addresses a major source of inaccuracy with the trace-driven approach that is generally relied upon in multi-component simulation.

Graphics processors are already ubiquitous on commodity machines; as heterogeneous computing grows, it becomes crucial for a simulator to efficiently take advantage of the new parallelization model available in heterogeneous computing. I demonstrate a network simu-

lator running on a GPU, and show how reciprocal abstraction enables concurrent simulation with a core simulator without heavy communication overhead.

The techniques I present in this thesis enable fast and accurate simulation of complex architectures with hundreds of cores. This thesis has both improved upon existing parallel simulation techniques and introduced a new parallelization model for efficient computer architecture research.

APPENDIX A

WORKLOADS

Table 7: Benchmarks used in this work, drawn from the SPLASH-2 [79] and PARSEC [6] parallel benchmark suites

Suite	Name	Description
SPLASH-2	barnes	N-body particle simulation
	fft	Fast Fourier Transform kernel
	lu.ncont	Matrix factorization into lower / upper triangular matrices
	ocean.cont	Ocean current simulation
	water.nsq	Simulation of a system of water molecules
PARSEC	blackscholes	Option pricing using partial differential equations
	canneal	Chip design route minimization using simulated annealing
	dedup	Data stream compression using deduplication
	fluidanimate	Fluid animation using smoothed particle hydrodynamics
	swaptions	Swaption pricing using the Monte Carlo simulation

Table 8: Synthetic Traffic Patterns

Pattern	Destination function description
bitcomp	invert bits
shuffle	shift bits left with wrap-around
transpose	switch row and column

APPENDIX B

SIMULATORS

In this appendix chapter, the simulators referenced in this thesis are listd and briefly described.

- Simplescalar [3] - Simplescalar is a serial core simulator. It functionally simulates user-level code and performs detailed simulation of an out-of-order core and cache hierarchy.
- Simics [45] - Simics is a serial multicore simulator. It functionally models both user and system-level code. Simics supports detailed cycle-level modeling of hardware modules.
- M5 [8] - M5 is a serial multicore simulator. It functionally models both user and system-level code. M5 supports detailed cycle-level modeling of hardware modules.
- Gems [46] - The gems toolset contains various detailed hardware models. Opal is a detailed processor model; ruby is a detailed memory system model; and Garnet is a detailed NoC model.
- Gem5 [7] - Gem5 is a serial multicore simulator. It functionally models both user and system-level code. In addition, GEM5 uses the GEMS toolset [46] for detailed modeling.
- Graphite [48] - Graphite is a parallel multicore simulator, which supports loose barrier and random-pair synchronization. It can run on distributed hosts. It relies on binary instrumentation for user-level functional modeling. Graphite models in-order cores and uses abstract modeling for contention-based hardware structures.
- Sniper [9] - Sniper is a parallel multicore simulator, which supports loose barrier synchronization. It uses binary instrumentation for user-level functional modeling. Sniper relies

on abstract models for out-of-order core modeling and for contention-based hardware structures such as the on-chip network. Further details can be found in Section 2.4.2.

- ZSim [67] - ZSim is a parallel multicore simulator, with two-phase synchronization. In the first phase, functional simulation is loosely-synchronized via barriers. In the second phase, detailed simulation uses conservative PDES to order events. ZSim relies on binary instrumentation for user-level functional modeling. It uses abstract models for contention-based hardware structures such as the on-chip network.
- SST [65] - The Structural Simulation Toolkit (SST) is a simulation framework which targets simulation of large-scale High Performance Computing systems at varying abstraction levels. It utilizes conservative PDES (see Section 2.4.4) to synchronize threads.
- Manifold [76] - The Manifold simulator is a parallel cycle-level architectural simulator built on top of SST [65]. Like SST, Manifold supports synchronization using conservative PDES. However, for faster simulation, Manifold also supports loose barrier synchronization.
- Hornet [43] - Hornet is a parallel network-on-chip simulator, which supports loose barrier synchronization. Further details can be found in Section 2.4.3.
- FIST [57] - FIST is a network-on-chip simulator. It uses abstract modeling to estimate latencies, and is trained on a detailed model. FIST can be run serially on a CPU or in parallel on an FPGA.

BIBLIOGRAPHY

[1] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42, April 2009.

[2] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, January 2009.

[3] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] Erik Berg and Erik Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *ISPASS*, pages 20–27. IEEE Computer Society, 2004.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[8] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[9] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[10] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, April 1981.

[11] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, Sept 1979.

[12] Jainwei Chen, L. Kumar Dabbiru, D. Wong, M. Annavaram, and Michel Dubois. Adaptive and speculative slack simulations of cmps on cmps. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 523–534, Dec 2010.

[13] Jianwei Chen, Murali Annavaram, and Michel Dubois. Slacksim: a platform for parallel simulations of cmps on cmps. 37(2):20–29, 2009.

[14] Matthew Chidester and Alan George. Parallel simulation of chip-multiprocessor architectures. *ACM Trans. Model. Comput. Simul.*, 12(3):176–200, July 2002.

[15] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Sangyeun Cho, Socrates Demetriades, Shayne Evans, Lei Jin, Hyunjin Lee, Kiyeon Lee, and Michael Moeng. Tpts: A novel framework for very fast manycore processor architecture simulation. In *ICPP*, pages 446–453. IEEE Computer Society, 2008.

[17] NVIDIA Corporation. Cuda c programming guide. Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[18] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. Gtw: a time warp system for shared memory multiprocessors. In *Simulation Conference Proceedings, 1994. Winter*, pages 1332–1339. IEEE, 1994.

[19] Braulio Adriano De Mello and Flávio Rech Wagner. A standardized co-simulation backbone. In *SoC Design Methodologies*, pages 181–192. Springer, 2002.

[20] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. Technical Report 0901, January 2009.

[21] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 350–361, Washington, DC, USA, 2004. IEEE Computer Society.

[22] D. Eklov and E. Hagersten. Statstack: Efficient modeling of lru caches. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65, March 2010.

[23] Sander De Pestel Stijn Eyerman and Lieven Eeckhout. Micro-architecture independent branch behavior characterization. 2015.

[24] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):3:1–3:37, May 2009.

[25] Ayose Falcon, Paolo Faraboschi, and Daniel Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 22–31. IEEE, 2008.

[26] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.

[27] Richard M Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 46–53. IEEE Computer Society, 1999.

[28] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, February 2010.

[29] Jan LM Hensen. A comparison of coupled and de-coupled solutions for temperature and air flow in a building. *ASHRAE transactions*, 105(2):962–969, 1999.

[30] Fabiano Hessel, P. LeMarrec, Carlos A. Valderrama, Mohamed Romdhani, and Ahmed Amine Jerraya. Mci- multilanguage distributed co- simulation tool. In Franz J. Rammig, editor, *DIPES*, volume 155 of *IFIP Conference Proceedings*, pages 191–202. Kluwer, 1998.

[31] Colin J Ihrig, Rami Melhem, and Alex K Jones. Automated modeling and emulation of interconnect designs for many-core chip multiprocessors. 2010.

[32] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 195–206, New York, NY, USA, 2006. ACM.

[33] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[34] Nan Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D.E. Shaw, J. Kim, and W.J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 86–96, April 2013.

[35] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 338–349, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Acm Sigplan Notices*, volume 37, pages 211–222. ACM, 2002.

[37] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology*, 2012.

[38] A.J. KleinOsowski and D.J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7–7, January 2002.

[39] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 185–194, New York, NY, USA, 2006. ACM.

[40] Benjamin C. Lee, Jamison D. Collins, Hong Wang 0003, and David Brooks. Cpr: Composable performance regression for scalable multiprocessor models. In *MICRO*, pages 270–281. IEEE Computer Society, 2008.

[41] Kiyeon Lee and Sangyeun Cho. In-n-out: Reproducing out-of-order superscalar processor behavior from reduced in-order traces. In *MASCOTS*, pages 126–135. IEEE, 2011.

[42] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, pages 181–190, 1999.

[43] Mieszko Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, Christopher W. Fletcher, Omer Khan, and Srinivas Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *ISPASS*, pages 175–185. IEEE Computer Society, 2011.

[44] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[45] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[46] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[47] Peter Martini, Markus Ruemekasten, and Jens Tölle. Tolerant synchronization for distributed simulations of interconnected computer networks. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 138–138. IEEE Computer Society, 1997.

[48] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In Matthew T. Jacob, Chita R. Das, and Pradip Bose, editors, *HPCA*, pages 1–12. IEEE Computer Society, 2010.

[49] Michael Moeng, Sangyeun Cho, and Rami Melhem. Scalable multi-cache simulation using gpus. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 159–167, July 2011.

[50] Michael Moeng, Alex Jones, and Rami Melhem. Reciprocal abstraction for computer architecture co-simulation. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE, 2015.

[51] Michael Moeng, Rami Melhem, and Alex Jones. Weighted-tuple synchronization for parallel architecture simulators. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*. IEEE, 2014.

[52] Shubhendu Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Husslederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin wind tunnel ii: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, 1997.

[53] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM.

[54] Sebastien Nussbaum and James E. Smith. Statistical simulation of symmetric multiprocessor systems. In *Proceedings of the 35th Annual Simulation Symposium*, SS '02, pages 89–97, Washington, DC, USA, 2002. IEEE Computer Society.

[55] U.Y. Ogras and R. Marculescu. Analytical router modeling for networks-on-chip performance analysis. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[56] Mark Oskin, Frederic T. Chong, and Matthew Farrens. Hls: Combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 71–82, New York, NY, USA, 2000. ACM.

[57] Michael Papamichael, James C. Hoe, and Onur Mutlu. Fist: A fast, lightweight, fpga-friendly packet latency estimator for noc modeling in full-system simulations. In *NOCS*, pages 137–144. IEEE Computer Society, 2011.

[58] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: a full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055. ACM, 2011.

[59] David A. Patterson. Ramp: research accelerator for multiple processors - a community vision for a shared experimental parallel hw/sw platform. In *ISPASS*, page 1. IEEE Computer Society, 2006.

[60] Michael Pellauer, Michael Adler, Michel A. Kinsy, Angshuman Parashar, and Joel S. Emer. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *HPCA*, pages 406–417. IEEE Computer Society, 2011.

[61] Kalyan S. Perumalla. Parallel and distributed simulation: Traditional techniques and recent advances. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 84–95. Winter Simulation Conference, 2006.

[62] J Porras, J Ikonen, and J Harju. Applying a modified chandy-misra algorithm to the distributed simulation of a cellular network. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 188–188. IEEE Computer Society, 1998.

[63] Shivani Raghav, Martino Ruggiero, David Atienza, Christian Pinto, Andrea Marongiu, and Luca Benini. Scalable instruction set simulator for thousand-core architectures running on gpgpus. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France*, pages 459–466, 2010.

[64] Dhananjai Madhava Rao, Narayanan V Thondugulam, Radharamanan Radhakrishnan, and Philip A Wilsey. Unsynchronized parallel discrete event simulation. In *Proceedings of the 30th conference on Winter simulation*, pages 1563–1570. IEEE Computer Society Press, 1998.

[65] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, R Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. The

structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[66] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[67] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)*, June 2013.

[68] Daeho Seo, Akif Ali, Won-Taek Lim, Nauman Rafique, and Mithuna Thottethodi. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 432–443. IEEE Computer Society, 2005.

[69] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.

[70] Daniel J. Sorin, Vijay S. Pai, Sarita V. Adve, Mary K. Vernon, and David A. Wood. Analytic evaluation of shared-memory systems with ilp processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 380–391, Washington, DC, USA, 1998. IEEE Computer Society.

[71] Standard performance evaluation corporation. http://www.specbench.org.

[72] Narayanan V Thondugulam, D Madhava Rao, Radharamanan Radhakrishnan, and Philip A Wilsey. Relaxing causal constraints in pdes. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 696–700. IEEE, 1999.

[73] R. Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 15–23, Dec 2001.

[74] Marija Trcka, Michael Wetter, and Jan Hensen. Comparison of co-simulation approaches for building and hvac/r system simulation. In *Proceedings of the International IBPSA Conference, Beijing, China*, 2007.

[75] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Comput. Surv.*, 29(2):128–170, June 1997.

[76] Jun Wang, Jesse Beu, Rishiraj Bheda, Tom Conte, Zhenjiang Dong, Chad Kersey, Mitchelle Rasquinha, George Riley, William Song, He Xiao, et al. Manifold: A parallel simulation framework for multicore systems. In *Performance Analysis of Systems and*

Software (ISPASS), 2014 IEEE International Symposium on, pages 106–115. IEEE, 2014.

[77] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C. Hoe. Simulation sampling with live-points. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pages 2–12, March 2006.

[78] Thomas F Wenisch, Roland E Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. Simflex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[79] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[80] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95. IEEE, 2003.

[81] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization. In *IPDPS*, pages 1–12. IEEE, 2010.

[82] Joshua J Yi, Lieven Eeckhout, David J Lilja, Brad Calder, Lizy Kurian John, and James E Smith. The future of simulation: A field of dreams. *Computer*, 39(11):22–29, 2006.